On Exploiting Symbolic Execution to Improve the Analysis of RAT Samples with angr

Serena Lucca, Christophe Crochet, Charles-Henry Bertrand Van Ouytsel, and Axel Legay

INGI, ICTEAM, Universite Catholique de Louvain, Place Sainte Barbe 2, LG05.02,01, 1348 Louvain-La-Neuve, Belgium
{firstname.surname}@uclouvain.be

Abstract. This article presents new contributions for Remote Access Trojan (RAT) analysis using symbolic execution techniques. The first part of the article identifies the challenges in the application of such an analysis, as well as the procedures put in place to address these challenges. The second part of the article presents a practical analysis of samples from known RAT families with the help of the SEMA toolchain.

Keywords: Symbolic Execution, Malware Analysis, Reverse Engineering, Remote Access Trojans

1 Introduction

According to Dataprot [16], in 2022, 560 000 new pieces of malware were detected daily. The increasing number of critical infrastructures relying on digital resources exposes society more and more to malware-related threats. It is thus important to develop techniques to detect and classify malware. Scanning a file for specific signatures common to malware provides a means of malware detection. Signatures help identify malware and classify them into malware families.

In the past, signatures have been represented by strings and byte sequences. Initial malware detection only required the *static* identification of these elements directly in the file, which could be completed without executing the malware (see [33]). Malware writers have easily circumvented this static approach by offering obfuscation techniques [27]. Tools like Ghidra [23] allow for an improved understanding of the functioning of a binary by reverse-engineering. However, static analysis is no longer sufficient for malware detection [3]. Contrary to the static approach, *dynamic analysis* consists in executing a binary in a controlled or test environment (e.g., a 'sandbox'). This form of execution makes it possible to reveal and monitor the behavior of the malware without harm to the host system. For example, such an approach can identify the malware signature as a succession of system calls [4, 11], monitor the malware's behavior, and detect how the malware would adversely interact with the host system. Unfortunately, malware can often detect that it is running in test environments. When this happens, the malware hides its behavior so that it is (falsely) considered benign [6].

Dynamic analysis is equally ineffective when analyzing malware designed to interact with the network environment. In this case, malware detection requires either access to the network (which is dangerous) or imitation of the behavior of the network in the protected execution environment [18] (which can be prohibitively resource-intensive). This situation is particularly problematic when analyzing Remote Access Trojan (RAT) malware [35] given most RAT malware is designed to be deployed in a network environment.

Performing analyses manually takes significant time but can offer detailed information about the behavior and impact of the malware. Of course, automated analyses are faster and scale better [36]. Unfortunately, this form of analysis could miss malware information. For example, an automated analysis often focuses on one execution path occurring in a specific environment. Moreover, malware developers are continuously developing new ways to evade detection tools [2].

Symbolic analysis [17,31] solves most of the previously identified issues. Symbolic analysis works with a symbolic mathematical representation of the different values that each variable can take during a piece of malware's execution. This approach permits multiple explorations of various sets of paths at each execution step. Procedures testing the safety properties of very complex systems first promoted symbolic analysis [17]. Along with learning algorithms, malware detection processes now widely use symbolic analysis [22,26,30]. Symbolic analysis allows learning algorithms to create more general signatures thanks to a better exploration of the different behaviors of a piece of malware. Analysts can apply symbolic analysis, often with other tools, to observe, analyze, and understand new malware families [10,25].

In [7], the authors use an extension of *angr* to analyze a RAT of the Enfal family. They demonstrate that it is possible to use symbolic analysis to discover all the commands supported by the RAT and to identify the behavior associated with these commands. The authors point out that obtaining this information is less time-consuming when symbolic analysis is deployed in comparison to a manual static analysis. However, their approach focuses on a single binary. The authors of [7] do not generalize their approach or apply it to different malware from different families. Another issue unaddressed by [7], is the need to automatically generate actionable reports of what is observed during their analysis. Because symbolic analysis can explore all the commands of the RAT, automated reporting would facilitate the quick comparison of different malware. Automated reporting could also aid in more efficiently analyzing the evolution of a malware family over time. In [12], the authors complement their previous work with the reconstruction of a C2 server for a Remote Access Trojan which can then be used to perform in-vivo analysis. Their approach uses symbolic execution on the full software stack to record interesting traces in the command processing loop, considerably increasing execution time. Their approach also requires an analyst.

Our work addresses these issues. This paper presents a generalization of symbolic analysis techniques, accounting for the specific workflow of a RAT. For example, we explore how to manage the command processing loop and competently handle time-consuming functions and external calls. We also propose how

to implement our approach in SEMA [8], a toolchain for malware analysis based on *angr*. Our approach includes, among other solutions, creating a database of time-consuming functions, implementing more than 100 SimProcedures to tackle the external system calls, and producing a report that focuses on Indicators of Compromise (IoCs). Our implementations are publicly available [5]. We also present a detailed analysis of a sample from the Warzone family and provide a brief overview of our analysis of a MagicRAT sample. With this last analysis, we show how symbolic analysis can be combined with static analysis.

2 Background

This section reviews concepts used in our work. We start with symbolic execution, which is a technique used to collect behaviors of a program. We then present the tool SEMA (Symbolic Execution for Malware Analysis) to automatize malware analysis. Finally, we discuss Remote Access Trojan (RAT), malware that can be used to gain remote control of a target computer.

Symbolic execution allows for the collection of a compact representation of sets of behaviors of a binary without executing it. In contrast, a concrete execution follows only one of many possible paths. While a concrete execution directly updates the variables in memory, symbolic execution replaces variables with symbolic values and saves the logical formulas defining those symbolic values. Symbolic execution abstractly executes a binary, considering multiple inputs concurrently. When the execution flow meets a branch depending on a symbolic value, it forks its current state of execution and continues with two different paths. In many cases, symbolic execution has shown better results than static and dynamic analysis for malware analysis [9, 10].

There are many challenges associated with symbolic execution. If the approach aims to explore all the possible execution paths, the situation can degenerate. Symbolic loops can generate an exponential number of paths [24]. This leads to the well-known *path explosion problem*. Another issue is environmental interaction; it is often difficult to symbolically interpret the output of external (and therefore unknown) API calls according to their inputs, which leads to the appearance of over-approximation. This necessitates improving approach efficiency with the design of exploration methods and heuristics [30].

angr [31] is an open-source binary analysis tool that provides a symbolic execution framework. In *angr*, all information related to the current state of execution of a program are stored in a *SimState* and represented by *symbolic bitvectors* (BVS). Relationships between BVS are represented by constraints from which concrete values can be inferred. During the execution, applying instructions generates successor states from a *SimState*. SimProcedures manage external API calls. SimProcedures are function summaries that reflect the effects of the calls on the *SimState*.

SEMA (Symbolic Execution toolchain for Malware Analysis) [8, 9] is a new *angr*-based tool that detects and classifies malware via symbolic execution and

machine learning (see Figure 1). (1) The tool accepts ELF and PE executables. (2) *SEMA* takes binary files as input and performs symbolic execution with *angr*. (3) During the Symbolic Execution process, API calls and their corresponding arguments are collected to create a System Call Dependency Graph (SCDG) [26] that links calls based on their dependencies. The tool implements several SimProcedures that are useful for malware analysis. Many of these SimProcedures are not implemented in the standard *angr* framework. Others are also optimized to improve the concretization of symbolic values, mainly of strings. This speeds up the symbolic execution and makes the analysis of the traces more convenient. SCDGs can be seen as an abstraction of malware signatures. (4) Machine learning algorithms can use SCDGs to build models for malware detection and classification [8, 9]. The machine learning models rely on graph mining through gSpan [37], graph kernel and support vector machine, or deep learning models. In this paper, we are particularly interested in extending the capabilities of angr in *SEMA*. The tool implements useful heuristics for malware analysis, such as path prioritization strategies to improve code coverage [9], loop handling [30], optimized SMT solver strategies [30],... With these heuristics, we are able to discover new addresses of the malware, avoid infinite symbolic loops, and efficiently explore the program's state space. *SEMA* also offers new *angr* plugins to collect information about malware interactions with the system (e.g., accessed environment variables, windows registries).

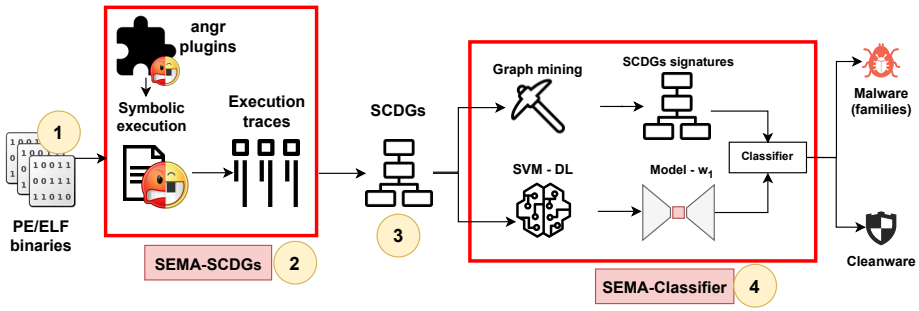

Fig. 1. SEMA

RAT analysis RAT malware offers remote access control which allows for multiple transgressions including launching DDoS attacks or stealing sensitive information. Having established persistence on the target computer, a RAT generally sets up a connection to the Command & Control (C2) server to communicate with an attacker. The RAT then enters a command processing loop where it will poll the server for instructions from the attacker. When the RAT receives a command from the attacker, it executes the corresponding function. The set of possible commands depends on the family and version of the RAT. Commands

can comprise keylogging, screen capture, execution of payloads, file ex-filtration, etc. RATs are out of the scope of most automatic techniques in malware analysis because they require network interaction to expose their malicious behavior.

3 Contributions to Symbolic Analysis of RATs

This section introduces improvements in symbolic execution to accelerate RAT analysis. Figure 2 shows a RAT generic workflow to present improvements.

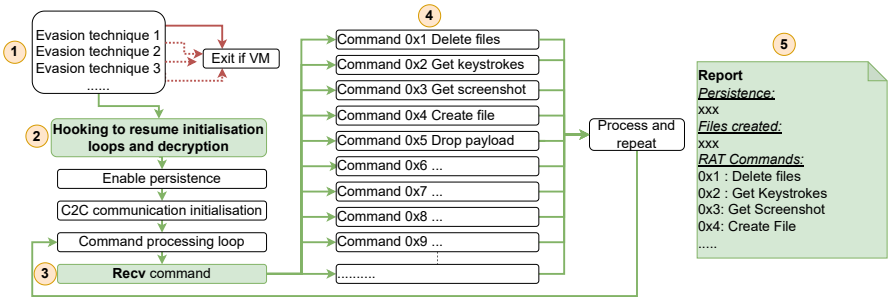

Fig. 2. Workflow of the RAT with our improvements

First, the RAT usually tries to detect if it is running in a monitored environment (e.g., a 'sandbox'). If the RAT determines this to be the case, then it may deploy evasion techniques. Each technique deployed will lead to the creation of a new symbolic path (1). Depending on the environment's reaction, the RAT may terminate its execution, generate a benign behavior, or continue its malicious execution. If the RAT continues malicious execution, it executes several procedures related to the initialization of the memory and some integrity checks (2). These procedures include, for example, the initialization of the allocated virtual memory to zero with a loop, or a CRC32 computation on the whole binary. These procedures are generally present in all variants of the same RAT family [14] and will not contain malicious actions useful in the identification or analysis of new RATs. They can also hinder execution efficiency. As a result, these procedures should be circumvented before the application of symbolic analysis to new candidates. The RAT will then establish persistence by modifying specific Windows configuration files. For example, many RATs add a path to the registry key `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`. After this, the RAT will initialize a connection with its C2 server and wait for commands (3). Managing the interaction with the C2 server is challenging. Ideally, all the commands need to be explored by the symbolic exploration engine to discover all the capabilities of the RAT. In addition, it is useful to understand the logic behind the different commands (i.e., which command triggers which behavior)

when analyzing the recorded network traffic. The SimProcedures of the external calls present in the Commands (4) should be provided to the symbolic analyzer.

General methodology Our approach includes manually analyzing a RAT sample and identifying the challenges for its family. We then tackle challenges with subsequently mentioned techniques that, once successful, are generic enough to be generalized across samples.

Explore of the binary Several strategies exist to explore symbolic execution paths efficiently. This includes breadth-first search and depth-first search. In [9], the authors proposed a depth-first search procedure (CDFS) that explores paths containing new instruction addresses. This algorithm is very effective when exploring the command processing loop (3). Each new command of the processing loop corresponds to a new path with addresses not yet visited. This exploration method thus helps efficiently visit all possible commands between steps (3) and (4) of Figure 2. This technique also helps with evasion techniques. As seen in Figure 2, evasion techniques often reuse the same address if they detect a Virtual Machine (1). A focus on new addresses leads to the malicious behavior.

Handle time-consuming functions Performing symbolic analysis requires using a mathematical model that represents sets of executions. At each step, a given instruction is applied to symbolic values represented by sets of constraints. This computation takes more time than the dynamic execution, which applies the instruction directly on concrete values [15, 38]. In some situations, such as loops, the repetition of identical instructions severely impacts the computation time. Our approach should manage such situations. As an example, consider a function whose objective is to copy part of the memory from one location to another. The code extracted from the binary by Ghidra is given in Listing 1.1. We note that the more the size of the area under replication increases, the more repetitively the code of the loop is executed. Avoiding this repetitive process in the symbolic analysis is preferable to reduce time consumption.

```
int copy_mem(int dest_addr,undefined *src_addr,int length){
  int offset;
  if (length != 0) {
    offset = dest_addr - (int)src_addr;
    do {
      src_addr[offset] = *src_addr;
      src_addr = src_addr + 1;
      length = length - 1;
    } while (length != 0);
  }
  return dest_addr;
}
```

Listing 1.1. Exemple of function managing memory

```
def copy_data(state):
    return_addr = state.stack_pop()
    length = state.stack_pop()
    src_addr = state.stack_pop()
    dest_addr = state.stack_pop()
    data = state.memory.load(src_addr, length)
    state.memory.store(dest_addr, data)
    state.stack_push(dest_addr)
    state.stack_push(src_addr)
    state.stack_push(length)
    state.stack_push(return_address)
    return
```

Listing 1.2. Hook that replaces the time-consuming function

To address this problem, we manually identify the time-consuming functions in one sample from a family. To do this, we monitor the symbolic execution and identify the addresses where the execution stalls. We then manually extract the behavior of each of these functions (e.g., with Ghidra) and create an equivalent Python function. For example, the Python code in Listing 1.2 is equivalent to the function in Listing 1.1. We verify that these models behave similarly to the actual functions by testing them on different inputs and comparing the outputs. Creating these Python models manually is tedious and only applicable to small and unobfuscated functions; it should be automated and generalized to all types of time-consuming functions in future work. When analyzing a new sample from a previously evaluated family, our approach deploys a pattern-matching algorithm to identify problematic functions found with manual identification in other samples from the family, performing a replacement to avoid running a symbolic execution on these functions.

Handle external calls and simulate interaction Thanks to SimProcedures (see Section 2), our approach manages external calls to reflect their environmental modifications (4). SimProcedures provide more visibility and control over the inputs and outputs of the API calls [21]. For example, if a system call receives a pointer to a buffer as an argument, its content is automatically retrieved from memory and saved for future analysis. SimProcedures also support the restriction of return values, helping to prioritize interesting paths and discard unwanted paths (e.g., paths related to successful evasion techniques (1) are automatically avoided because they return values not related to any sandbox). Finally, SimProcedures permit us to simulate network interactions. In Windows binaries, network interactions and message communication happen through system calls such as `recv` or `InternetReadFile`. Contrary to dynamic execution, our approach does not require crafting concrete messages to trigger malware behaviors. Instead, our approach returns a symbolic message from the SimProcedures related to those system calls, permitting the automatic discovery of any behavior that depends on the message.

Produce reports Initially, SEMA stores all system calls and their arguments found in the various execution traces in a json file. As already highlighted in previous research [7], execution traces from symbolic execution are difficult for analysts to interpret. That is why our improvements include the production of a report that gathers structured information about the behavior of the analyzed malware. More precisely, the report focuses on Indicators of Compromise (IoCs) such as modifications in registry keys, creation of new files/processes, connection to specific IP addresses, etc. Analysts share these IoCs with the infosec community to improve knowledge of malware behavior, incident response, and remediation strategies [34]. In our tool, this information is automatically extracted from the aforementioned json file, selected based on a list of relevant system calls [19]. Since the focus of this paper is RAT malware, a report on the different commands available in the RAT is created as well (5). By recording which received message (3) corresponds to which execution path (4), it is possible to understand the relationship between the message and the associated behavior (i.e., all system calls that occurred during the execution trace between the receipt of two messages). For instance, if command 0x1 triggers the calls `CreateFileA` and `WriteFile`, we can infer that command 0x1 corresponds to the creation of a file on the host.

4 Experiments

In this section, we show how to use techniques introduced in Section 3 to obtain a detailed analysis of different RATs families. We propose a detailed analysis of the Warzone RAT and briefly discuss another family of interest. Before diving into the analyses, we quickly show how our work improves on [7]. The experiments presented in this section are performed on a computer with a 12th Gen Intel Core i7-1255U × 12 and 16GB RAM running Ubuntu 20.04.5.

4.1 The Enfal family from [7]

In [7], the authors present a manual analysis of a sample from the Enfal RAT malware family via symbolic execution. The sample can be found on MalwareBazaar [1] with the md5 7296d00d1ecfd150b7811bdb010f3e58. The approach used in that paper relies on the capacity of an analyst to identify the portion of code that requires analysis. In practice, the analysis focuses on the extraction of the communication protocol used by the malware. This analysis produces a report that lists the different execution traces and the resulting API calls. From this report, the authors extract the different commands that the RAT features. In their work, the authors mention that their approach does not provide a clear summary of the execution. The authors also claim that a generalization of their approach may be hindered if each sample needs a different setup. Discovering the right setup for each sample requires the intervention of an analyst, which takes time. In this context, the authors raise the question of how one can minimize the manual intervention required to analyze a malware sample. In our approach, we

propose to mitigate the problems exposed in [7]. Our tool automatically parses the execution traces to find predefined IoCs related to the malware's behavior (e.g., registry modification, file creation, network communication, etc.). We also devise a number of automatic optimizations to improve the symbolic execution and minimize manual interventions. We develop techniques that, once applied to specific malware, can be automatically applied to similar samples from the same family.

4.2 Warzone RAT

The Warzone malware is written in C++ and targets Windows users. The format of the file is a 32-bit portable executable (PE). For simplicity, we focus on the analysis of a sample that can be found on MalwareBazaar [1] with the MD5 93c5434350e0f5dc53a88202ee48e531. We successfully analyzed 35 samples of this family due to improvements made for the first sample.

General workflow At the beginning of the execution, Warzone goes through several time-consuming functions. Because these functions can considerably slow down symbolic execution, they must be identified. Once the initialization is complete, Warzone tries to become persistent by copying itself to the folder `C:\Users\User\AppData\Roaming`. This path is then added to the registry key `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`. The execution continues with the privilege escalation. This process is performed differently depending on the host's Windows version. The malware next ensures that it will escape antivirus detection. Warzone achieves this by adding its name to the exclusion list of Windows Defender. The malware then connects with its C2 server and enters the command processing loop where it receives and executes the commands sent by the attacker. There are more than 20 different commands ranging from keylogging to shell execution. The communication between the victim and the C2 server is encrypted with RC4 and the password `"warzone160\x00"`.

Improved exploration Our first contribution is to compare the effect of different symbolic exploration strategies. We observe that the CDFS strategy from [9] greatly improves code coverage compared to classical DFS/BFS strategies. This technique enables us to visit all the possible commands that the RAT features. In comparison, we observe that with a DFS strategy, the execution will always go depth-first and therefore get stuck in the infinite loop (4). When using the BFS method, the execution is slower because it explores many redundant paths. For example, the exploration will create two branches to execute the privilege escalation depending on the OS version of the host. After escalating privileges, the rest of the execution of each branch will use the same instructions at the same addresses. In practice, the situation repeats itself and creates an exponential blow-up which impacts the execution time and the instructions that are visited. The CDFS strategy [9] detects this situation and limits its exploration to a single branch. Table 1 shows a comparison. In terms of analysis quality, the technique from [9] identifies all the different commands and finds more system

calls than BFS or DFS within the same time limit.

Method	# of instructions visited	# of blocks visited
CDFS	10264	2145
BFS	4141	1001
DFS	2828	644

Table 1. Code coverage for three different exploration method

User defined hooks Several hooks have been implemented to avoid the symbolic exploration of time-consuming functions. Functions for Warzone that require management include data processing functions (as seen in Listing 1.1), cryptographic routines (MurmurHash), and other time-consuming functions. These functions are generic and are reused in other samples from the family. Hence, our pattern-matching algorithm automatically finds the problematic functions in other samples from the family and replaces them with previously created hooks. Figure 3 represents the time (in logarithmic scale) spent in different parts of the execution. The experiment was performed on three equivalent runs of 600 seconds. In the first run, which does not apply hooks, most of the execution time is spent in cryptographic functions. In the second run, we see that the execution gets trapped in the data processing functions if corresponding hooks are not implemented. Finally, when all hooks are implemented, we see that the time-consuming functions no longer have an impact on the execution.

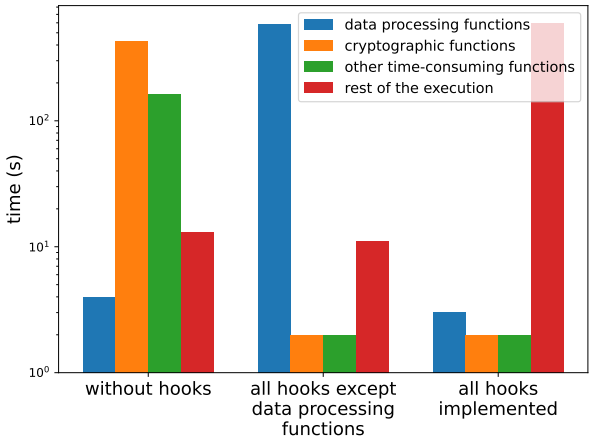

Fig. 3. Execution time with and without hooks

Interaction and SimProcedures We have implemented all the SimProcedures that correspond to the system calls made by the RAT during its execution. This represents a substantial effort since more than 100 SimProcedures have

been considered. As an example, the code provided in Listing 1.3 shows the SimProcedure for the function `GetVersionExA`. This function is used to determine the OS version, which is important in the selection of the privilege escalation technique. The implementation is based on the information found in the Win32 API documentation. For instance, the variable `MajorVersion` can take different values between 5 and 10 depending on the OS. In the SimProcedure, this is represented by creating a symbolic variable and applying constraints. Our strategy in implementing SimProcedures is to limit the symbolic inputs and outputs to what is specified in the Win32 API. We discard only the paths that would lead to an error in real execution. In this way, we mitigate the path explosion problem while preserving all paths that would be dynamically feasible.

```
class GetVersionExA(angr.SimProcedure):
  def run(self, addr):
    MajorVersion = claripy.BVS("MajorVersion",32)
    self.state.solver.add(MajorVersion >= 5)
    self.state.solver.add(MajorVersion <= 10)
    self.state.memory.store(addr+0x4, MajorVersion)
    MinorVersion = claripy.BVS("MinorVersion",32)
    self.state.solver.add(MinorVersion >= 0)
    self.state.solver.add(MinorVersion <= 3)
    self.state.memory.store(addr+0x8, MinorVersion)
    BuildNumber = claripy.BVS("BuildNumber",32)
    self.state.memory.store(addr+0xc, BuildNumber)
    PlatformId = claripy.BVV(2,32)
    self.state.memory.store(addr+0x10, PlatformId)
    return 0x1
```

Listing 1.3. SimProcedure of `GetVersionExA`

The `recv` SimProcedure plays an important role in the execution of all the RAT malware that communicates through this function. During analysis, a symbolic buffer is created to represent the buffer returned by the `recv` function. This is done in a way that permits the retrieval of the constraints that were applied to the buffer. This helps deduce the concrete value of the buffer for each execution trace and create a report on the commands that the RAT can receive.

Report extracted Each execution automatically produces two reports, one for the commands and one for the IoCs. For each execution trace, the commands report produces the command message that triggered this trace and the following system calls. As an example, the first part of Listing 1.4 shows a piece of the report that represents an execution trace. It corresponds to the command that uninstalls the malware. This report supports the comparison of different samples from the same family. If we analyze samples spread over a large time period, we might see a growth in the number of commands in recent versus older malware. The IoC report is divided into several categories. Each category represents a potentially incriminating behavior. Listing 1.4 provides an excerpt of the IoC report from the Warzone analysis. In Category "Network activity", we observe that the address of the C2 server is "`rtyui.nerdpol.ovh`". This category also

shows all the system calls that are used to communicate on the network. The next category lists all the activities related to the registers. This is strategic information, as the registers are often used by malware to create persistence. The "Files" category lists all the system calls that concern files and directories. In this example, we see that Warzone asks for a special folder to which it will copy itself later. The categories "Processes" and "Command Line" retrieve information on the processes created by the malware. Here we see that Warzone creates the processes `sdclt.exe` and `cmd.exe`, used to escalate privileges in Windows.

```
RAT Commands:
* 0x9123b422d33a244e6018673
- RegDeleteKeyW
- GetModuleFileNameA
- CreateProcessA
- CloseHandle
- ExitProcess
[...]
Network activity:
- inet_addr(rtyui.nerdpol.ovh)
- ...
Registers:
- RegOpenKeyExW(Software\Microsoft\Windows\CurrentVersion\
    Explorer\ZU6FTFTS7G)
- RegCreateKeyExA(Software\Classes\Folder\shell\open\command)
- ...
Files:
- SHGetSpecialFolderPathW(C:\Users\USERNAME\AppData\Roaming)
- ...
Processes:
- CreateProcessW(C:\Windows\System32\cmd.exe)
- ...
Command line:
- ShellExecuteW(open, C:\Windows\System32\sdclt.exe)
- ...
```

Listing 1.4. Report on discovered commands and IoCs

4.3 MagicRAT

We analyzed a sample of the *MagicRAT* family, a malware attributed to the North Korean Lazarus group. The sample can be found on MalwareBazaar with the MD5: b4c9b903dfd18bd67a3824b0109f955b. Cisco Talos [32] recently discovered it. MagicRAT (written in C++, compiled as a 64-bit PE file) uses the widespread and trusted QT graphical framework. This significantly increases the size and complexity of the binary. Since the binary of the RAT is complex and large, we performed the analysis with a combination of Ghidra and SEMA.

General workflow With symbolic analysis, we found that the malware starts with an initialization phase of its parameters. Then it creates and hides its configuration inside a file called `\ProgramData \WindowsSoftwareToolkit\visual.`

`1991-06.com.microsoft_sd .kit` from the legit `QSettings` class. The configuration is composed of three strings that start with the common prefix `"LR02DPt22R"`. During the execution, this prefix is removed from each string. The remaining part of the string is then decoded to obtain the C2 URLs (`"LR02DPt22R<url_i_encoded_in_base64>"`). By inspecting the result of the symbolic execution, we were able to retrieve the IP from the string. This allows for threat hunting and forensic analysis of these sources. We then used Ghidra to identify places where suspicious strings (e.g., `"cmd.exe /c bcdedit"`) are exploited. This speeds up workflow deduction. With our analysis, we observe that MagicRAT uses the task scheduler to execute itself at 10:30 a.m. each day. It also hides in a fake "Onedrive" shortcut in the startup folder of the victim. Its command processing loop is simple; it mainly manipulates victims' files (rename, delete, and move). MagicRAT can also execute terminal commands on the victim to collect information which is sent to the attacker in a file named `"zero_dump.mix"`. Finally, the sample has a self-deletion procedure contained in a `".bat"`. We now show the implementation contributions that allow us to achieve this result.

User defined hooks The QT framework requires specific CPU features for the execution which can be checked with the assembly instruction "cpuid". *angr* does not handle this instruction. Thus, we implement a special hook called `"CPUIDHook"`. This is an advantage when compared to dynamic analysis because our approach works independently of the real CPU used.

Improved exploration. The QT framework contains code that is not relevant for symbolic analysis but can hinder effectiveness. The logging process of QT induces an exploration of benign behavior associated with different environmental variables. Using Ghidra, we identify 17 QT environment variables associated with such optional behavior. By setting values for these variables, we avoid examining these behaviors and improve the efficiency of the analysis that would otherwise time out before discovery of the malicious behavior.

Interaction and SimProcedures All API calls present in the command loop are implemented in SEMA. This includes Windows structures containing file information such as `_BY_HANDLE_FILE_ INFORMATION`, which contains file metadata, or `VS_VERSIONINFO`, which gives the version of the file. We have also implemented file mapping. This typically starts with a call to `CreateFileMappingW` to create a handle for the mapping of an input file. `MapViewOfFile` then maps the content of the file to the memory region.

5 Conclusion and Future Work

We have demonstrated how symbolic analysis can help understand the workflow of RATs by proposing new heuristics implemented in SEMA and by using a combination of manual and automatic actions. While our work significantly improves the analysis of challenging malware such as RATs, more remains to be

done. Our approach reduces the cost of symbolic execution by identifying and replacing bottleneck functions. Our pattern-matching approach, while efficient, is prone to syntactic obfuscation. This process could improve with more reliance on obfuscation-resilient semantic functionality identification as in [28]. The bottleneck functions could also be automatically identified with the use of machine learning as in [20]. Monitoring specific symbolic values (e.g., values related to interaction with the network) could also enhance exploration techniques [13, 29]. Finally, we will integrate our contributions into the machine learning malware detection process introduced in [26] and implemented in SEMA. In particular, we intend to improve detection and classification of RAT malware families on a larger scale.

Acknowledgments. This research is supported by the Walloon region's CyberExcellence program (Grant #2110186).

References

1. Abuse.ch: Malwarebazaar (2023), https://bazaar.abuse.ch/
2. Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D.: Malware dynamic analysis evasion techniques: A survey. ACM Computing Surveys (CSUR) **52**(6), 1–28 (2019)
3. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Network and Distributed Systems Security (NDSS) Symposium 2020 (2020)
4. Amer, E., Zelinka, I.: A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence. Computers & Security **92**, 101760 (2020)
5. Anonymous, A.: SEMA for RAT. https://github.com/AnonymousSEMA/SEMA-RAT
6. Avllazagaj, E., Zhu, Z., Bilge, L., Balzarotti, D., Dumitras, T.: When malware changed its mind: An empirical study of variable program behaviors in the real world. In: USENIX Security Symposium. pp. 3487–3504 (2021)
7. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C.: Assisting malware analysis with symbolic execution: A case study. In: Cyber Security Cryptography and Machine Learning: First International Conference, CSCML 2017, Beer-Sheva, Israel, June 29-30, 2017, Proceedings 1. pp. 171–188. Springer (2017)
8. Bertrand Van Ouytsel, C.H., Crochet, C., Legay, A.: Tool paper-sema: Symbolic execution toolchain for malware analysis. In: 17th International Conference on Risks and Security of Internet and Systems (2022)
9. Bertrand Van Ouytsel, C., Legay, A.: Malware analysis with symbolic execution and graph kernel. In: Reiser, H.P., Kyas, M. (eds.) Secure IT Systems - 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30-December 2, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13700, pp. 292–310. Springer (2022)
10. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: An overview of malware detection and evasion techniques. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation.

Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11244, pp. 565–586. Springer (2018)
11. Blokhin, K., Saxe, J., Mentis, D.: Malware similarity identification using call graph based system call subsequence features. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops. pp. 6–10. IEEE (2013)
12. Borzacchiello, L., Coppa, E., D'Elia, D.C., Demetrescu, C.: Reconstructing c2 servers for remote access trojans with symbolic execution. In: Cyber Security Cryptography and Machine Learning: Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27–28, 2019, Proceedings 3. pp. 121–140. Springer (2019)
13. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. Botnet Detection: Countering the Largest Security Threat pp. 65–88 (2008)
14. Calleja, A., Tapiador, J., Caballero, J.: The malsource dataset: Quantifying complexity and code reuse in malware development. IEEE Transactions on Information Forensics and Security **14**(12), 3175–3190 (2018)
15. Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I.: {SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2531–2548 (2022)
16. Dataprot: A Not-So-Common Cold: Malware Statistics in 2022 (2023), https://dataprot.net/statistics/malware-statistics/
17. Godefroid, P.: Test generation using symbolic execution. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India. LIPIcs, vol. 18, pp. 24–33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012)
18. Gorecki, C., Freiling, F.C., Kührer, M., Holz, T.: Trumanbox: Improving dynamic malware analysis by emulating the internet. In: SSS. pp. 208–222. Springer (2011)
19. HackTricks: Common api used in malware (2023), https://book.hacktricks.xyz/reversing-and-exploiting/common-api-used-in-malware
20. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R.: Function representations for binary similarity. IEEE Transactions on Dependable and Secure Computing **19**(4), 2259–2273 (2021)
21. Microsoft: Programming reference for the win32 api (2023), https://learn.microsoft.com/en-us/windows/win32/api/
22. Namani, N., Khan, A.: Symbolic execution based feature extraction for detection of malware. In: 2020 5th International Conference on Computing, Communication and Security (ICCCS). pp. 1–6. IEEE (2020)
23. NSA: Ghidra (2023), https://ghidra-sre.org/
24. Obdržálek, J., Trtík, M.: Efficient loop navigation for symbolic execution. In: Automated Technology for Verification and Analysis: 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings 9. pp. 453–462. Springer (2011)
25. Park, K., Sahin, B., Chen, Y., Zhao, J., Downing, E., Hu, H., Lee, W.: Identifying behavior dispatchers for malware analysis. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 759–773 (2021)
26. Said, N.B., Biondi, F., Bontchev, V., Decourbe, O., Given-Wilson, T., Legay, A., Quilbeuf, J.: Detection of mirai by syntactic and behavioral analysis. In: Ghosh, S.,

Natella, R., Cukic, B., Poston, R.S., Laranjeiro, N. (eds.) 29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018. pp. 224–235. IEEE Computer Society (2018)
27. Schrittwieser, S., Katzenbeisser, S.: Code obfuscation against static and dynamic reverse engineering. In: Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13. pp. 270–284. Springer (2011)
28. Schrittwieser, S., Kochberger, P., Pucher, M., Lawitschka, C., König, P., Weippl, E.R.: Obfuscation-resilient semantic functionality identification through program simulation. In: Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings. pp. 273–291. Springer (2023)
29. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE symposium on Security and privacy. pp. 317–331. IEEE (2010)
30. Sebastio, S., Baranov, E., Biondi, F., Decourbe, O., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Optimizing symbolic execution for malware behavior classification. Comput. Secur. **93**, 101775 (2020)
31. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). pp. 138–157. IEEE (2016)
32. Talos, C.: Magicrat: Lazarus' latest gateway into victim networks (2022), https://blog.talosintelligence.com/lazarus-magicrat/
33. Team, Y.: Yararules (2023), https://github.com/Yara-Rules/rules
34. TrendMicro: Indicators of compromise (2023), https://www.trendmicro.com/vinfo/us/security/definition/indicators-of-compromise
35. Valeros, V., Garcia, S.: Growth and commoditization of remote access trojans. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 454–462. IEEE (2020)
36. Vasilescu, M., Gheorghe, L., Tapus, N.: Practical malware analysis based on sandboxing. In: 2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference. pp. 1–6. IEEE (2014)
37. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: 2002 IEEE International Conference on Data Mining, 2002. pp. 721–724. IEEE (2002)
38. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 745–761 (2018)