Obfuscation Resilient Behavior Based IDS based on Colored Petri Nets

Arnur G. Tokhtabayev, Victor A. Skormin and Andrey M. Dolgikh
Center for Advanced Information Technologies, Binghamton University
{atokhta1, vskormin, adolgik1}@binghamton.edu

Abstract – Behavior based intrusion detection became the only effective solution against modern malware that usually employ binary morphism to avoid conventional anti-viruses. Dynamic behavior-based IDSs (BBIDS) depends on three interrelated factors: signature expressiveness, behavioral obfuscation and run-time efficiency. Signature expressiveness determines the success of IDS in detecting multiple realizations of the same malware. Behavioral obfuscation/metamorphism is an emerging threat perceived as a common feature of the future information attacks. Signature matching efficiency determines the scalability of dynamic BBIDS. This paper addresses these aspects of BBIDS development. To achieve higher signature expressiveness, we present a new approach for formal specification of the malicious functionalities based on activity diagrams (AD) defined in an abstract domain. So-called abstract functional objects are introduced for creating highly generic specifications yet preserving the discriminatory properties. Resultant AD would incorporate multiple realizations of the specified functionality hence increasing semantics and expressiveness of the signature. Possible behavioral obfuscation techniques, inter-process and intra-process, that can compromise existing BBIDS are analyzed and classified. To mitigate obfuscations, we propose the augmentation (generalization) of otherwise obfuscation prone specifications into more generic, obfuscation resilient specifications. To achieve high signature matching efficiency, we propose the utilization of colored Petri nets (CPN) for recognizing functionalities at the system call level. We suggest the incorporation of the information flows into CPN to achieve a fine-grained recognition. Finally, we propose a procedure that translates AD into CPN that recognize these AD in the system call domain enriched with information flow data. The proposed techniques have been implemented in a prototype IDS and evaluated on dozens of malware and hundreds of legitimate programs. The experimental results indicate low false positives and negatives, as well as low execution overhead and negligible overhead penalty due to anti-obfuscation generalization.

Index Terms — IDS, Colored Petri Nets, System calls, Dynamic behavior detection, Behavior obfuscation, Behavior metamorphism

1 Introduction

Computer networks, being a critical component of the national infrastructure, are continuously subjected to information attacks. Most devastating attacks are perpetrated by the deployment of self-replicating malicious software propagating through different media and using multiple attack vectors. Recent information attacks demonstrate steady increase in professionalism and sophistication of newly deployed malware. To avoid signature-based detection by most commercial intrusion detection systems (IDS), modern malicious software is at least polymorphic and sometimes metamorphic. Fortunately, IDS utilizing behavioral signatures to match malware activity rather than its binary structure are immune to binary morphism.

While behavior-based IDSs (BBIDS) have obvious advantages, they could suffer from three interrelated problems: signature expressiveness, behavioral obfuscation and run-time signature matching efficiency. Signature expressiveness determines the success of IDS in detecting new realizations of the same malware. Since most malware incidents are derivatives of some original malware, a successful signature must capture invariant generic features of the entire malware family. At the same time, the signature should be expressive enough to reflect the most possible malware realizations. Behavioral obfuscation is an emerging threat that, given the extensive development of BBIDS, is expected to become a necessary and trivial feature of future information attacks [36].

Behavior of a program can be viewed as a manifestation of the functionalities implemented in the program. A particular functionality is malicious if it performs some specific activities intended for adversarial purposes. Discovering a malicious functionality in any software qualifies it as a malware. Hence, detection of malicious functionalities becomes crucial and sufficient for confident malware detection.

We developed a novel system call domain IDS that addresses existing and future challenges of BBIDS. To achieve higher signature expressiveness, we proposed to specify the functionalities of interest, specifically malicious ones, by activity diagrams (AD) in terms of both standard system objects and abstract behavioral constructs named functional operations. The utilization of functional objects and operations
provides the necessary level of generalization yet preserves discriminatory properties of the specification. As a result, such an AD would incorporate multiple realizations of the specified functionality hence increasing semantics and expressiveness of the signature.

We investigated possible approaches to behavioral obfuscation including inter-process (multipartite) techniques. To mitigate obfuscation, we propose automatic generalization of AD specifications. We developed a set of generalization algorithms that automatically augment signatures making them resilient to several behavioral obfuscation techniques, such as object relocation and multipartite activity.

Finally, we developed a procedure capable of automatic conversion of activity diagrams into system call domain Colored Petri nets (CPN), intended for run-time recognition of the specified functionalities in IDS. Our experiments showed that a CPN are highly dependable and efficient for recognizing specified functionalities in the flow of system calls as well as utilized data.

The system architecture of the proposed IDS is shown in Figure 1. At the learning phase, an expert designs activity diagrams (AD) representing known malicious functionalities. The Specification Generalizer module automatically augments original AD making them more generic and resilient to obfuscations. The CPN Constructor generates low-level and high-level CPN by processing relevant AD. The low-level CPN recognize individual subsystem-level object operations in the system call domain thus aggregating system call information for processing at the higher level. The high-level CPN recognize specified functionalities in the domain of object operations. While simulating CPN, Recognizer accesses the information flow tracer to feed data dependencies for particular transitions of CPN.

As shown in Figure 1, at the detection phase, Object Operation Recognizer receives system calls and utilizes low-level CPN to identify subsystem object manipulations. Functionality Recognizer utilizes the high-level CPN to assemble object operations into particular functionalities.

The contributions of our paper are as follows.

*Increasing signature expressiveness and simplifying the process of signature specification:*
  * Formal functionality specifications using AD defined at the abstract object level. Each specification allows for capturing multiple alternative realizations of functionality
  * Separation of the specification domain (abstract OS objects) from the detection domain (system calls). Abstract specification domain allows an expert to concentrate on conceptual realizations of a functionality omitting certain implementation details. The detection domain allows for efficient functionality detection in the system call flow by executing respective CPN obtained from the specification
  * Automation of the IDS signature generation process. It includes computer aided AD specification design, automatic AD generalization, AD visualization and finally automatic translation of the AD to a CPN used as a signature in the intrusion detector (Functionality Recognizer)

*Mitigation of possible behavioral obfuscations:*
  * Analysis and classification of possible behavioral obfuscation techniques
  * Automatic generalization of functionality specifications thus making them invulnerable to behavioral obfuscation

*Achieving high efficiency of signature matching:*
  * Automatic translation of an AD specification into a CPN that recognizes the functionality in the system call domain
  * Prototype of information flow tracing engine (implemented in IDA Debug)

To demonstrate our approach we implemented it in a prototype IDS and tested by detecting several malicious functionalities employed by network worms and bots, including self-replication engines and various malicious payloads.
The rest of the paper is presented as follows. In Section 2, we formalize the functionality specification. In Section 3, we address certain behavior obfuscation techniques via generalization. Section 4 presents specification recognition through CP-nets. In Section 5, we describe our approach for information flow tracing. Section 6 is dedicated to implementation of our methods in the prototype IDS. Sections 7 and 8 feature experimental evaluation of the prototype IDS.

2 Functionality specification

2.1 Formalization of the specification

Before formalizing functionality specification, let us view functionality from OS perspective as presented below. The MS Windows OS provides system resources and services to processes through executive objects maintained in the Windows Kernel. In order to access a particular resource or service, a process creates a corresponding object such as a file, process, thread, memory section, etc [1]. Every object has its own set of operations which are exported to user mode processes through system services (system calls)\(^1\). In the user mode, such system calls are invoked directly or more conveniently through subsystem API functions.

Processes invoke API functions or system calls to perform object operations (manipulations)\(^2\) that complete some semantically distinct actions, such as writing data to a file or sending data to a specified IP address. Consequently, we define individual functionality as a combination of actions that achieve a certain high-level objective. It is important to understand the difference between a functionality and behavior. The behavior of a process is what the process does at the particular stage, while the functionality determines semantic goals of the process. In other words, behavior simply manifests the realization of functionality. As a result, the major limitation of the existing behavior-based specifications is that they fail when dealing with multiple realizations of the same functionality. This motivated us to develop a novel specification free from this shortcoming.

### Table 1 “Remote shell” realizing

<table>
<thead>
<tr>
<th>Bind Shell realization</th>
<th>Reverse shell realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. h_In=CreateNamedPipe( dwOpenMode=PIPE_ACCESS_INBOUND SecurityAttributes.bInheritHandle=TRUE);</td>
<td>1. s=socket();</td>
</tr>
<tr>
<td>2. h_Out=CreateNamedPipe( dwOpenMode=PIPE_ACCESS_OUTBOUND SecurityAttributes.bInheritHandle=TRUE);</td>
<td>2. connect(s, sockaddr.s_addr=Attacker_IP sockaddr.sin_port=Attecker_port);</td>
</tr>
<tr>
<td>3. ConnectNamedPipe(h_In);</td>
<td>3. CreateProcess(“cmd.exe”, bInheritHandles = TRUE, STARTUPINFO.dwFlags = STARTF_USESTDHAND, STARTUPINFO.hStdInput= h_In, STARTUPINFO.hStdOutput= h_Out);</td>
</tr>
<tr>
<td>4. ConnectNamedPipe(h_Out);</td>
<td>STARTUPINFO.hStdOutput= hSock);</td>
</tr>
<tr>
<td>5. CreateProcess(“cmd.exe”, bInheritHandles = TRUE, STARTUPINFO.dwFlags = STARTF_USESTDHAND, STARTUPINFO.hStdInput= s, STARTUPINFO.hStdOutput= s);</td>
<td></td>
</tr>
</tbody>
</table>

Note that processes may utilize both user level objects exported by the windows environment subsystem, such as Socket, Memory Mapping etc., and kernel level objects exported by the object manager, such as File,

---

\(^1\) In Unix based systems such services are called system calls, while in Windows they are called executive system services. Hereafter, we stick to system call term.

\(^2\) In this paper, we use terms “Operation” and “Manipulation” interchangeably, because both of the terms are used extensively in the literature.
Named Pipe, Memory Section etc. For user level objects, we consider subsystem level operations exported by API functions of subsystem libraries (such as kernel32.dll, ws2_32.dll). However, each subsystem level object is based on kernel level executive object.

Table 1 features a simple functionality, “Remote Shell”. This functionality creates a backdoor allowing an attacker to remotely execute system commands. Remote Shell has at least two possible realizations: Bind Shell and Reverse Shell. Both realizations create the “cmd.exe” process with input and output buffers (hStdInput, sStdOutput) being set to a connected port. The difference between these realizations is that the Bind Shell accepts a connection via named pipes and Reverse Shell connects to the attacker via sockets. Both realizations invoke CreateProcess API with specific flags that allow for using socket/pipe handle as an input/output. This makes the command interpreter listen to incoming commands and execute them.

Consider requirements for specifying functionality. Based on the above example and [2], [3], we formulate the following requirements for the functional specifications:

1. **The specification must define the control flow for object operations. It must support conditional branching and concurrent execution.**

   *Conditional branching* allows for specifying alternative realizations that may utilize different objects and operations; however, they achieve the same goal determining the functionality. For instance, two realizations in Table 1 utilize two different objects: “Named Pipe” and “Socket”.

   *Concurrent execution* allows for specifying independent object manipulation sessions, that could be executed in any order. However, the sequence of depended operations must remain intact within the session. For instance in Bind Shell realization (Table 1, left side), there are two independent operation sessions: create inbound pipe and get it connected (APIs 1 and 3); create outbound pipe and get it connected (APIs 2 and 4). Since these two sessions are independent, the API functions (1, 2 and 3, 4) could be invoked in any order, as long as API 3 follows API 1 and API 4 follows API 2.

2. **The specification must define data/information flow among object operations.** This requirement allows for specifying how output attributes of operations become the inputs of consequent manipulations. An attribute data flow determines the discriminatory power of the specification. For instance, with “Remote Shell” functionality, we have to show that “Process” object is created with STARTUPINFO.hStdInput and STARTUPINFO.hStdOutput attributes being set to the socket handle.

   We should point out that data may be passed by value (data flow) or by information (information flow) [27]. Information flow between source and destination attributes indicates that value of the destination is a transformation of the value of the source.

In addition to the above two requirements, we introduce the third one that overcomes certain limitations in [3] related to multi-processes activity.

3. **The specification shall not be constrained to the context of one process.** This allows for specifying and relating operations of different processes. In fact, this allows for specifying inter-process functionalities.

The specification must offer enough expressive power and convenient graphical notation to simplify its designing. Many specifications provide graphical notations: state diagrams/machines, simple flowcharts, and workflow diagrams. However, state machine is a uniprocess model that does not meet Requirement 3 and cannot directly express data flows as per Requirement 2. Flowcharts do not support concurrent execution and do not meet Requirement 1. However, a workflow diagram such as UML Activity Diagram (AD) can generally meet the above requirements. Consider the formalization of an AD in terms of an OS object operations for the purpose of functionality representation.

A basic UML AD is a semantically weighted directed graph:

$$G = (\text{Nodes, Arcs, Guards})$$

where Nodes = State ∪ Pseudo. The set State contains state nodes that represent executed activities; it also contains Initial and Final nodes that represent the beginning and end of the process. The set Pseudo comprises pseudo state nodes that control the execution flow. Pseudo state nodes include decision/merge (for conditional flow branching), and fork/join (for concurrent flow execution). Set Guards contains guard expressions (for conditional branching) that represent the semantic weight of the corresponding edges.

Consider the “Remote Shell” functionality in Table 1. Figure 2 depicts an AD of the functionality in the graphical (left side) and analytical (right side) forms. According to UML 2.x standards, the graphical notation displays decision/merge nodes (a, d) as diamonds, and fork/join nodes (b, c) as bars. We also assigned a sequential index to each node for explanatory purposes.

The AD contains seven state nodes and four pseudo-state nodes. The state node set includes four created
object instances (Nodes 1, 3, 5, 7) and three operations (Nodes 2, 4, 6) on these objects. The pseudo-state nodes determine control flow of the functionality. The decision node “a” starts two alternative realizations of the functionality. The left branch (Nodes 1-2) represents the first step of the Reverse Shell realization, while the right branch (Nodes 3-6) represents the first step of the Bind Shell realization. Node 7 is the common step for both realizations. Note that the first step of the Bind Shell realization (Nodes 3-6) has two independent sessions (Nodes 3-4 and 5-6).

The graphical notation is convenient for an expert designing a specification. However, the analytical representation is crucial for automatic processing of the specification. The analytical form of “Remote Shell” functionality shown in Figure 2 (right side) is very consistent with the UML AD formalism (1). The only modification we made is the inclusion of \( Vars \) component that represents a set of local variables. Below we explain all components of our formalism in the generic fashion (the detailed formalism of the AD specification is given in Appendix A).

The functionality specification is defined as an AD tuple:

\[
F = (\text{Nodes, Arcs, Assign, Vars})
\]  

where \( \text{Nodes} \) is a multi-set defined in line 2 (Fig. 2). It consists of State and Pseudo nodes. As defined in line 3, there are two types of State nodes: Instances and Manipulations. Each Instance node represents an object created and operated in the context of the functionality, and its attributes. Line 7 shows the set of Instance nodes for “Remote Shell” functionality. The set includes four nodes corresponding to the following objects: socket (Node 1), two named pipes (Nodes 3 and 5) and the process (Node 7). Line 7 indicates that each object is defined with the attributes used in corresponding API creating the object. Each Manipulation node represents an object manipulation with appropriate parameters. Line 8 shows Manipulations set for “Remote Shell”. The set includes three manipulations: connect socket (Node 2), and connect named pipes (Nodes 4 and 6).

**Arcs** is a set of directed arcs connecting the AD nodes. As defined in line 4, the arcs could be of two

---

1 In the brackets, we show the node index as presented in the graphical form.
types: Handle flow and Control flow. Handle flow arcs correspond to the execution flow with handle inheritance. A handle arc indicates that the destination operation (node) utilizes the same object handle as a source operation and is executed right after the source. In other words, the handle arc passes the handle of the source node to the destination node. For instance in Figure 2, the arc between Nodes 1 and 2 is a handle arc. It passes the handle of the socket created in Node 1 to the connect operation in Node 2 (this indicates that socket from Node 1 is operated at Node 2). To distinguish the handle arcs from control arcs, we display the handle arcs beginning with rectangle.

Control Flow arcs define the control flow without handle inheritance. An arc from this set indicates the execution order and does not imply any data binding (via handle or attribute). For instance, the arc between Node “d” and Node 7 is a control arc that does not transfer any handles, simply showing that Node 7 should be executed right after the two sessions (Nodes 1-2 and Nodes 3-6). In order to transfer data between nodes, variables are used. For instance, the pipes or socket handles are transferred to Node 7 through variables V1, V2.

**Assign** is a function that binds variable assignment expressions to corresponding arcs. Line 11 shows the definition of such a function. The function indicates that variable V1 is assigned with handle that was utilized in Node 2 or Node 3.

**Vars** is a set of local variables that are used to define data flow. Utilization of local variables with assignment expression allows for specifying data flow between object operations (the second specification requirement). To specify an information flow, an expert should use transformation notation T() as depicted in Appendix A. Note that it is possible to define informational dependency explicitly as well as implicitly. Explicit information flow implies specifying formal mapping between source and destination. Implicit flow does not specify any mapping simply stating that the destination value should at least partially depend on the source value.

To address the third specification requirement, each State node is assigned a unique index of the process that performs the manipulation represented by the node. Note that in Figure 2, “Remote Shell” is defined as an intra-process functionality, hence all the operations are invoked by the same process, and every State node of the AD has the same process index which is 1 (see line 2).

From Figure 2, one could see that the graphical representation of an AD is much more revealing than the analytical representation; however, both representations are formally identical. In the rest of the paper, the graphical representation will be used for explanatory purposes, while the analytical representation will be used in specification processing algorithms.

### 2.2 Specification abstraction

The detection success of our system highly depends on how comprehensive the specification is. If an expert misses a functionality realization, the system would be prone to false negatives. Each specification must be as generic as possible. It should abstract certain implementation details enabling experts to concentrate only on conceptual realizations. This is accomplished by the introduction of so-called functional objects that represent some complex but rather standard OS functionalities/mechanisms such as Inter-Process Communication (IPC), File Download etc. Note, a functional object abstracts several alternative realizations of the particular OS functionality by encapsulating the necessary Windows objects utilized in these realizations. Each functional object has a set of operations representing certain high-level activities. While specifying an AD, experts may create and manipulate functional objects as ordinary Windows objects. Table 2 shows the set of functional objects facilitating data transfer. This set is just an example and is far from being complete. However, it demonstrates the expressiveness of our approach which is the ability of operating abstract objects by other objects. For brevity, we discuss only few functional objects, the purpose of the rest of them corresponds to their names.

Object “GenericFile” abstracts file access operations; it encapsulates both the “file” object and “file mapping” object. Object “RemoteIPC” represents IPC resource for inter-host data transfer. It abstracts three alternative IPC mechanisms: socket, named pipe and mailslot. The “RemoteIPC” object exports the following operations: Create, Wait, Recv and Send. AD of these operations are shown in Appendix B. Some operations have input and output attributes. For instance, operation “RemoteIPC Create” requires two input attributes: Endpoint class (either server or client), EndPoint ID (host IP and Port for socket and name for pipe/mailslot). The operation returns two outputs: EndPoint type (socket, pipe or mailslot) and a handle value of the corresponding object. Operation Wait(h) waits for incoming connection to the newly created IPC endpoint with handle h.
Note that from the expert’s perspective, the utilization of such functional objects is transparent. For instance, when using RemoteRPC in a specification, the expert should not make any assumptions on how a malware will perform IPC, through a socket, pipe or mailslot. Such a transparency is best exemplified by FileTransfer operations. Table 2 indicates that FileTransfer operations are based on sheer functional objects such as GenericFile and RemoteIPC. This demonstrates the generalization power of the proposed specification formalism. Armed with such functional objects, an expert can build quite generic specifications yet preserving discriminatory properties that would leave little room for detection evasion.

### 3 Mitigating behavior obfuscations

The discriminatory power of a behavior signature (functionality specification) defined by an expert could be quite subjective and may exclude some of the realizations of the functionality. In addition, an attacker may perform some sort of behavioral obfuscation to evade detection. To address this issue, we developed a set of algorithms that automatically generalize (augment) the specification of the functionality. In the rest of the section, we first discuss possible behavioral obfuscations then we introduce the generalization algorithm and show how it addresses various obfuscation techniques.

#### 3.1 Behavior obfuscation techniques

By utilizing functional objects, experts may specify most of the realizations of the functionality. Then it would be difficult for an attacker to discover yet another conceptually different realization utilizing different Windows objects. However, to evade detection, an attacker does not have to implement a completely new realization. He may simply obfuscate a known realization in such a way that it would break the specification. We distinguish inter-process and intra-process approaches to obfuscate a realization without affecting the functionality. Inter-process obfuscations utilize multiple interrelated processes that at high level, jointly perform a particular malicious functionality. Intra-process obfuscation locally alters a realization of the functionality while preserving its behavioral semantics.

First, consider possible inter-process approaches to behavioral obfuscation.

1. **Utilization of legitimate third party utilities to perform required activity.** A malicious process may run third party utilities to execute some important tasks that may be a part of the functionality. In this way, the process executes the functionality without performing some key object manipulations involved in the task. For instance, a file virus usually searches for executables using “FindFirstFile” and “FindNextFile” API. Instead, the virus may utilize system command interpreter (e.g. “cmd.exe”) to retrieve a list of executable files in a folder and then access the files one by one.

2. **Distribution of the functionality among several processes a.k.a. multipartite approach.** A multipartite malware consist of several agents that perform coordinated activity to achieve a common goal. Such malware can distribute a malicious functionality among several processes by injecting its code to active benign processes or by creating new ones. Then the combined activity of these processes...
will perform an inter-process malicious functionality. A real life example of such a malware is a KeyLogger which is described in the next section. Another example is a File Virus that consists of two processes. The first process opens an executable file and passes the file handle to the second process. Then the second process attaches the code of the first process to the opened victim file. It could be seen that none of the processes performs a typical malicious functionality individually: the second process does not open the victim file and does not inject its code, while the first process is replicated into the victim file without performing write or self-access operations.

Now let us consider intra-process obfuscation approaches.

3. **Object relocation and duplication** Since a functionality may be constrained by particular object name (e.g. file name), an attacker changes the name of the object before manipulating it. For instance, an attacker can copy, rename or move a file before manipulating it. In addition, a malware may duplicate object handle in the middle of the manipulation sequence to break system call binding. Sometimes, an attacker may access objects through symbolic links instead of handles.

4. **Non-direct object manipulation** It is achieved by specific, low-level system tricks such as utilization of non-trivial OS resources that allow for accessing objects whether in non-trivial way or through a “middleman” object. For instance, an attacker can create reparse points or access files by their streams. He also may add an alternative path to a target file through relinking system calls. Such activities are performed only through Kernel objects using system calls.

3.2 **Obfuscation mitigation through generalization**

In the system architecture presented in Figure 1, “Specification Generilizer” module addresses the above obfuscation techniques. Effectively, this module attempts to fill up expert’s experience/attention gap, thus alleviating limitations related to a human factor. The module applies a set of generalization algorithms that automatically augment a given AD to make it less prone to obfuscations. Herein, we propose the following generalization algorithms:

- **TraceFiles** – Augments the given AD with functionalities tracing the renaming and relocation of all files involved in the specification. This algorithm addresses the third obfuscation technique.
- **TraceHandles** – Augments the given AD with functionalities that trace object handle propagation among processes, which requires tracking handle duplication and IPC used for handle transfer. This addresses the first three obfuscation techniques.
- **TraceProcesses** – Augments the given AD with functionalities that track process generation, remote code injection and inter-process coordination. This involves detecting several realizations of code injection including remote thread based and remote hook based. The upgraded AD would be able to relate object manipulations performed by multiple processes. This algorithm mitigates the first and the second obfuscation techniques.

To address the forth obfuscation approach, one does not need any post-processing of the AD in the generalization stage. Instead, we can simply extend functional objects with necessary semantics that would trace low-level objects involved in the obfuscation. This results in the obfuscation being resolved at the stage of specification, rather than automatic post-generalization. Particularly, we add reparse points and file streams into “GenericFile” functional object.

While augmenting an AD, each of the generalization algorithms incorporates special functionalities, termed generalization functionalities, which trace certain activity involved in particular obfuscation. Table 3 describes the generalization functionalities, whose AD are given in Appendix C. As it could be seen in Table 3, functionalities maintain certain global variables which qualify the traced activities, e.g. generated processes, duplicated files or established IPC channels.

We describe each of the generalization algorithms based on their pseudo-code, where we utilize several **primitive functions** defined in Appendix D and generalization functionalities defined in Table 3.

**TraceFiles** pseudo-code is given below. The algorithm iterates over operations and instances presented in an AD (line 1). If an operation has “file name” as an argument (line 2), the procedure adds “FileRelocation” functionality to the AD (line 6). Note, the function $AttList(x)$ returns a list of attribute names for the input operation $x$.

While adding a parallel functionality is trivial, it is not obvious where to insert “FileRelocation” so that file tracing does not interfere with the rest of the original functionality. Thereby, we insert parallel flow with “FileRelocation” in the following way. If the target file name is a constant string, i.e. is independent from other operations of the AD, we start the parallel flow right after the initial node. If the file name is a variable,
we start the parallel flow right after the node where the variable is assigned for the last time. Lastly, we join “FileRelocation” parallel flow with the original AD right before the node performing the operation on the target file.

In the algorithm, the parallel flow with “FileRelocation” functionality is added by function $AddParallelFunc(Origin,New,Start,Merge)$ (see Appendix D). It adds an AD named $New$ to an AD named $Origin$ as a parallel flow that starts right after the node $Start$ and joins to the AD $Origin$ just before the node $Merge$. The node $Start$ is determined in lines 4 and 5. If the file name is a variable, the node $Start$ is defined through $GetAssignNode$ function (line 4). Function $GetAssignNode(x)$ returns the node which output arc has an assignment expression for variable $x$. Line 7 modifies the AD to make it consistent with the AD formalism (2) given in Section 3.

**Algorithm TraceFiles**

Input: AD - An activity diagram specification  
Output: Generalized AD

```
1. foreach Operation ∈ \{AD.Instances ∪ AD.Manipulations\} :
2.     if (IpFileName ∈ AttList(Operation)) :
3.         TargetFileName := GetAttributeValue(Operation,IpFileName);
4.     if (isVariable(TargetFileName)) :
5.         RelocStartNode = GetAssignNode(TargetFileName)
6.     else : RelocStartNode = AD.initial;
7.     AddParallelFunc(AD/FileRelocation(TargetFileName),RelocStartNode,Operation);
8.     SetAttributeValueExpression(Operation,IpFileName,IpFileName in FList["+TargetFileName +"]);
```

**Table 3 Generalization Functionalities (introduced by generalization algorithms)**

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Updated variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileRelocation</td>
<td>FList</td>
<td>Accepts a file name as input and updates a FList variable which a dictionary indexed by file names. Each element of FList is a list containing names of duplicates of the input file indexing the element. Such duplicates could be derived by copying, moving or renaming of the original input file or any of its duplicates.</td>
</tr>
<tr>
<td>ProcessGeneration</td>
<td>PList</td>
<td>These two functionalities trace process generation and inter-process code injection and constantly update global variable PList. PList is a list containing PID of the descendant (created or being injected) processes (up to given generation limit) originated from the initial process that starts AD.</td>
</tr>
<tr>
<td>CodeInjection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleDuplication</td>
<td>DupP DupH</td>
<td>Traces handle duplication and constantly updates two global variables: DupP and DupH. <strong>DupP</strong> is a dictionary indexed by value of the initial handle produced by object creation. Each element of the dictionary is a list containing PIDs of the processes possessing duplicate handles derived from the handle indexing the element. <strong>DupH</strong> is a two dimensional dictionary indexed by value of the original handle and PID. Each element of the dictionary is a list of handles possessed by indexing PID and derived from the indexing handle. For instance, $DupH[H1][P1]＝{H2,H3}$ means that handles $H2$, $H3$ derived from the original handle $H1$ and possessed by $P1$ process.</td>
</tr>
<tr>
<td>LocalIPC Establishment</td>
<td>IPC_P IPC_H</td>
<td>Traces Local IPC establishment and constantly updates two global variables IPC_P and IPC_H. <strong>IPC_P</strong> is a dictionary indexed by ID of the IPC. Each element of IPC_P is a list of PIDs of the processes that own endpoint handles (including duplicates) of the IPC with ID indexing the element. For instance, $IPC_P[\text{id1}]＝{\text{PID1, PID2, PID3}}$ means that IPC, identified by id1, has endpoints which handles are possessed by processes PID1, PID2 and PID3. Note that some IPC could serve as data share points, hence may have multiple endpoints, for instance a file or a shared memory. <strong>IPC_H</strong> is a two dimensional dictionary indexed by IPC ID and PID. Each element of the dictionary is a list of endpoint handles (including duplicates) that are possessed by indexing PID and shared by IPC with ID indexing the element. For example, $IPC_H[\text{id1}][\text{PID1}]＝{h1,h2,h3}$ means handles $h1,h2,h3$ represent endpoints of the IPC with id1 ID and possessed by process with PID1 ID.</td>
</tr>
</tbody>
</table>
TraceHandles pseudo-code is given below. In the code, line 1 introduces “HandleDuplication” functionality as a parallel flow to the original functionality. Lines 2-8 constitute a loop that iterates over all object instances of the AD so that for each instance, a new element in DupH dictionary is initialized with PID and Handle of the instance (lines 3, 4). This would allow “HandleDuplication” functionality to trace handle duplicates of the current object instance. Line 6 iterates over object operations performed on the current object instance. For each object operation, the algorithm redefines PID and Handle expressions to allow the operation to utilize any duplicated handle belonging to original object instance.

Algorithm TraceHandles
Input: AD - An activity diagram specification
Output: Generilized AD

1. AddParallelFunc (AD, HandleDuplication, AD.initial, AD.final);
2. foreach Object ∈ AD.Instances:
3.   SetAssignExpression (OutputArc (Object), "DupH[Handle][PID] = \{Handle\}’’);
4.   SetAssignExpression (OutputArc (Object), "DupP[Handle] = \{PID\}’’);
5.   HandleVarName = CreateNewVar (OutputArc (Object), "Handle’");
6.   foreach Operation ∈ GetObjectOperations (AD.Object):
7.     SetNodePIDExpression (Operation, "PID in DupP[\’\’+
8.     SetAttributeValueExpression (Operation, Handle, "Handle in DupH[\’\’+

TraceProcesses addresses the first and the second obfuscation methods. In the first obfuscation, a malware runs an external utility to perform some tasks. As a result, the external utility has to utilize the OS resources the same way as the malware. In other words, malware simply outsources its operations or functionalities to the utility. We can recognize the outsourced functionality in the utility’s behavior using our specifications. Consequently, in the specification some object manipulation sessions must have PID tag assigned to the PID of the utility. If the utility has started, we must record the PID of the utility process and assign the PID in the object operation sessions that is outsources.

From the above perspective, starting a utility to perform a part of the malicious functionality represents a multipartite approach. Hence, the first and the second obfuscation techniques should be addressed similarly: by tracing the functionality distribution among several processes. This requires tracking processes generated by the malware as well as processes to which malware injected its code (infected). Then we attribute object operations to the generated processes and infected processes.

TraceProcesses algorithm introduces “ProcessGeneration”, “CodeInjection” and “LocalIPCEstablishment” to the input AD. It also introduces IPC required for coordinating multipartite agents and/or communicating with the utility. To reduce false positive rate we additionally trace data transmission between processes that represents technical yet vital activity. For instance, a process retrieves (reads) data through an object, representing data source, and then this data or its informational dependency is transferred (written) through another objet, called data sink (see Table 4). Distributing this activity in such a way that one process would access a source object and another process would access a sink object requires using IPC responsible for data transmission from the source process to the sink process. Such distributed functionality in fact implements an inter-process information link between source and sink objects (recourses).

For the sake of clarity, in Table 4, we present OS/functional objects and their corresponding operations that could be used for data source and sink. Note that some objects share the same source/sink operations.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Source operation</th>
<th>Sink operation</th>
<th>Based on API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source operation</td>
<td>Sink Operation</td>
<td></td>
</tr>
<tr>
<td>File, Pipe, MailSlot</td>
<td>Read</td>
<td>Write</td>
<td>ReadFile.kernel32,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WriteFile.kernel32</td>
</tr>
<tr>
<td>Socket</td>
<td>Recv</td>
<td>Send</td>
<td>recv.ws2_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>send.ws2_32</td>
</tr>
<tr>
<td>Registry</td>
<td>ReadValue</td>
<td>WriteValue</td>
<td>RegGetValue.Advapi32.dll</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RegSetValueEx.Advapi32.dll</td>
</tr>
<tr>
<td>RemoteIPC, LocalIPC</td>
<td>Recv</td>
<td>Send</td>
<td></td>
</tr>
</tbody>
</table>
Below we give pseudo-code for TraceProcesses algorithm. Initially the algorithm introduces three functionalities (lines 2-4), LocalIPC Establishment, ProcessGeneration and CodeInjection (see Table 3). The functionalities are incorporated through AddParallelFunct function that adds a functionality to AD as a parallel flow to the entire original activity. In line 5, the algorithm iterates over all objects of the input AD and changes their PID assign label to “PID in PList”. This means that PID must belong to the global list PList that contains PIDs of the children processes derived from the original malicious process or its children (see Table 3).

Lines 6-20 constitute the main loop that locates for source-sink operations (Table 4) and introduces data transmission functionality between source and sink processes. Line 6 iterates over all operations of the AD. If an operation is a sink (line 7), the algorithm obtains writable buffer/"pBuffer" attribute value. If the pBuffer attribute value is a variable (checked in line 9), the algorithm obtains the node that assigns the variable (line 10). Such assigning node is viewed as a data source and data is transferred to the sink operation through SinkBuffer variable. Next, the algorithm introduces IPC between source process and sink process, to achieve this, the algorithm adds send and recv operations as a parallel flow (lines 10-14) between the node assigning the variable (source node) and the current operation writing the assigned variable (sink node). Lines 15-20 set PID and attribute expressions of the newly introduced recv node and the current sink operation node, so that to operation would belong to the same process.

To demonstrate our generalization algorithms, we tested them with a simple functionality that uploads a file through IPC. We utilized Visual Paradigm for UML software [24] to design the “File Upload” functionality, see Figure 3. Then we run the prototype of a Specification Generalizer module (Figure 1) that automatically generalized the designed AD of the functionality using all three algorithms. Figure 4 shows the AD of the augmented (generalized) functionality. Note that Visual Paradigm designer, with minor manual alignments, automatically produced both AD layouts of Figures 3 and 4. It could be seen that in our prototype, the entire process of AD generalization is completely automated, which includes computer aided AD design, automatic generalization, and finally visualization of the resultant AD.

```
Algorithm TraceProcesses
Input: AD - An activity diagram specification
Output: Generalized AD

1. Sinks = {Write, Send, WriteValue}
2. AddParallelFunct(AD, LocalIPC Establishment, AD.initial, AD.final);
3. AddParallelFunct(AD, ProcessGeneration(ThisPID), AD.initial, AD.final);
4. AddParallelFunct(AD, CodeInjection(ThisPID), AD.initial, AD.final);
5. foreach Object ∈ AD.Objects : SetNodePIDExpression(Object, "PID in PList");
6. foreach Oper ∈ AD.Operations :
7. if Oper ∈ Sinks :
8. SinkBuffer = GetAttributeValue(Oper, pBuffer);
9. if isVariable(SinkBuffer) :
10. AssignPIDVarName = CreateNewVar(Oper, BufferAssigningNode, "PID");
11. NewSendNode = AddParallelNode(AD, Send, [Object, AssignPIDVarName], pBuffer, SinkBuffer, Oper, BufferAssigningNode);
12. IPCIDVarName = CreateNewVar(Oper, NewSendNode, "ID");
13. NewRecvNode = AddNextNode(AD, Oper, NewSendNode);
14. SetNodePIDExpression(NewRecvNode, "PID in IPC_P[" + IPCIDVarName + "]");
15. SetAttributeValueExpression(NewRecvNode, Handle, "Handle in IPC_H[" + IPCIDVarName + "]([PID])");
16. RecvBufferVarName = CreateNewVar(Oper, NewRecvNode, pBuffer);
17. RecvPIDVarName = CreateNewVar(Oper, NewRecvNode, "PID");
18. SetNodePIDExpression(Oper, "PID := " + RecvPIDVarName);
19. SetAttributeValueExpression(Oper, pBuffer, "pBuffer := " + RecvBufferVarName);
20. 
```
As per example of Figure 3, one can see “File Upload” AD that has two independent sessions such as IPC establishment (Nodes 3-5) and reading a file (Nodes 1, 2). After establishing an IPC, a buffer is to be received through IPC and is to be written to the opened file. Node 6 represents some additional activity not related to data variables of the functionality. Node 7 sends the buffer with the file content via the established IPC. Note that in this AD, we utilized only functional objects and manipulations, hence covering most realizations of the “File Upload” functionality.

In the generalized AD (Figure 4), it could be seen that HandeDuplication, ProcessGeneration and IPCEstablishment functionalities are introduced in the original functionality as independent sessions. Since the name of the uploaded file is the input parameter, the Specification Generalization has introduced “File relocation” functionality right before the file open operation (Node 2). In Nodes 4 and 6, global handle dictionaries (DupH, DupP) are initiated with IPC object handle. HandleDuplication (Node 13) traces handle duplication and updates the dictionaries. Node 6 accepts connection to IPC server endpoint. One can see that in the process of generalization, TraceHandles algorithm has modified attributes of several operations: in Nodes 3 and 6, PID assignment was set to “PID in Dup[H1]” expression which means that PID of the operation must belong to the set of PID’s that posses the original handle H1.

Figure 3 File Upload AD
(AD is designed in and visualized by Visual Paradigm Suite)

Figure 4 Generalized File Upload AD
(AD is produced by Specification Generizer module and visualized by Visual Paradigm Suite)

TraceProcesses algorithm has introduced IPC in Nodes 12, 7 and 8 that transfers a buffer from the source process (reading the content of the file to be uploaded) to the sink process (sending the buffer to a
remote host). One can see that IPC nodes are introduced as an independent session from the other activity (Node 9).

Figure 4 indicates that generalized AD is structurally more complex than the original AD (Fig. 3). However, with respect to the number of state nodes it is comparable to the original. At the same time, the generalized AD addresses all three obfuscations presented above. This shows that proposed anti-obfuscation generalization causes rather acceptable complexity penalty.

As another example, we applied the above algorithms to generalize the AD presented in Figure 2. The generalized AD is shown in Figure 5. One can see that fork node “b” starts two sessions. The left session (Nodes 1-3) corresponds to the first steps of “Reverse Shell” and “Bind Shell” realizations. The right session is represented by one operation (Node 4), “FileRelocation” that traces “%systempath%\cmd.exe” file and outputs a list of files that descended from it.

“Remote Shell” realization is started in Node 1. It creates RemoteIPC object as a client that connects to the attacker host. “BindShell” realization creates RemoteIPC server in Node 2. RemoteIPC object handle is traced by “HandleDuplication” functionality (Node 6). Node 3 corresponds to accepting connection to the IPC endpoint. Nodes 1, 2 and 3 have PID index. Note that PID index is a part of formalism of AD presented in Section 3. Expression “PID in PList” means that PID of the process performing the operation must belong to the PList. Nodes 1, 2 and 3 represent inter-process part of the “Remote Shell” functionality. Such inter-process part along with Node 7 addresses obfuscation techniques 2 and 3. Indeed, nodes 1, 2 and 3 outsource IPC creation to other processes.

The final step of the “Remote Shell” is to run “cmd.exe”. Node 5 creates a process whose image belongs to the list of files originated from “cmd.exe”. Note that this FList was produced by “FileRelocation” operation (Node 4). Moreover, the process is created with standard input set to duplicate/original handle of the IPC endpoint, server or client.

Let us compare the generalized AD of “Remote Shell” functionality (Fig. 5) with the original AD (Fig. 2). The generic specification defines six different realizations against two of the original AD. All generic realizations are resilient to obfuscation techniques presented above. In spite of generalization, the structural complexity of the generalized AD is commensurable with the complexity of the original AD. In fact, the overhead imposed by generalization can be managed via algorithm parameters. For instance, generation threshold parameter in TraceProcesses to some degree determines the overhead of the tracing functionality. This demonstrates the effectiveness and flexibility of our approach.

Understandably, the more obfuscation techniques we address, the more complex the generalized specification is expected to be. However, the specification is not yet a recognition mechanism since it merely represents how the functionality is implemented in terms of object manipulations. Hence, efficiency of the recognition mechanism determines how many obfuscations we can address. We proposed highly efficient way to detect specified functionalities. The proposed recognition model is scalable enough to detect specifications with all obfuscations we are aware of.

4 Functionality recognition

As indicated in the system architecture (Fig. 1), the functionality recognition process consists of two stages. At the first stage, we recognize individual object manipulations by identifying their dedicated APIs in the system call domain. However, a manipulation may be performed through several alternative APIs
operating on the same Kernel objects. On the other hand, an API function may invoke several additional
minor system calls that are not critical for the manipulation implementation. Hence, only the essential,
semantically critical part of an API function should be recognized and attributed to the corresponding
manipulation. This recognition approach would be resistant to certain evasion techniques when the malware
do not invoke the entire API but only its critical system calls, thus only partially realizing the API yet
achieving the required manipulation.

At the second stage, we recognize functionalities through the identified object manipulations, i.e. APIs.
Note that system calls represent APIs and APIs represent functionalities in full consistence with the AD (2).
Hence, the same type of models could be employed for the recognition of subsystem object manipulations
and malicious functionalities. The selection of a particular recognition model must be justified with respect
to both expressive power and implementation efficiency (computational and memory complexity).

Justification of the recognition model with respect to expressive power. Consider the following simple
functionality that could be a part of a virus: “open all executables in a folder; do not access files until some
determined point in time; then check to see if all of them are ready for code injection; then if all the files
are ready, inject the code, otherwise close all of the opened files”. One realization of such functionality
could utilize the CreateFile, ReadFile (to read PE header) and WriteFile API (to inject code). Since this
functionality requires the synchronization of file reading and file writing, then the sequence of the API
invoked by the functionality would represent the following pattern:

\[
\text{CreateFile, CreateFile, ... CreateFile, ReadFile, ReadFile, ... ReadFile, WriteFile, WriteFile, ... WriteFile.}
\]

This pattern constitutes a formal language: 
\[ L = \{ \text{CreateFile}^i, \text{ReadFile}^i, \text{WriteFile}^i : i \in \mathbb{Z}^+ \} \].
According to Pumping lemma, this language is not context-free [4], but it can be generated by context-sensitive grammar.
We believe that context-sensitive grammar can express all functionalities presented in AD formalism (2).
Object parameters and handle values could be represented by a large non-terminal alphabet covering the
entire parameter space (i.e. all possible values of the parameter’s type). Hence, a functionality can be at least
recognized by Linear Bounded Automata (LBA) that is an accepter for a context-sensitive language [4].

Justification of the recognition model with respect to computational complexity.

According to [5], a LBA of size \( n \) can be simulated by a Place Transition Net (PT-net) of size \( O \left( n^2 \right) \).
Moreover, the LBA and equivalent PT-net would have identical time complexity for an acceptance problem.
A PT-net can be translated into an equivalent CPN in such a way that the structural complexity of the PT-net
(number of places) would be converted into the inscriptional complexity of the CPN (arc expressions). Since
a CPN would have far fewer places, we prefer using the CPN rather than a PT-net. Moreover, a CPN has an
advantage over a LBA in processing multiple instances of the operation chains (words): should a chain be
executed more than once, an LBA model accommodates extra states for each instance of the chain. In
contrast, a CPN represents an executed operation chain as one token residing in the corresponding place that
allows for processing of multiple chain instances with low overhead. Consequently, a CPN was chosen as a
recognition model.

A CPN could formally be defined as a tuple [6]:
\[
\text{CPN} = (S, P, T, A, N, C, G, E, I)
\]  
(3)
where: 
\( S \) – color set, 
\( P \) – set of places, 
\( T \) – set of transitions, 
\( A \) – set of arcs, 
\( N \) – node function, 
\( C \) – color function, 
\( G \) – guard function, 
\( E \) – arc expression function, 
\( I \) – initialization function.

Below, we formulate a CPN configuration that provides execution semantics for the AD specification
defined in (2). To recognize functionalities specified in AD, a CPN configuration must reflect objects and
manipulations. Moreover, we should recognize several distinct functionalities that may or may not have
common implementation patterns. Hence, CPN places must represent the following states: created objects,
object manipulations, pseudo states routing the control flow of ADs, and individual functionalities.

The above considerations indicate that the set of places of the CPN should consist of four disjoint sets:
\[
P = P_{\text{obj}} \cup P_{\text{manip}} \cup P_{\text{func}} \cup P_{\text{pseudo}}.
\]  
These disjoint sets determine the following four types of places:

**Object place** \( (P_{\text{obj}}) \) associated with a unique OS object. In this place, a token represents instance of the
object associated with the place. Such a token is defined as a tuple: a descriptor (handle) of the object
instance and a set of necessary object parameters. Hence, the color set of Object-places typically constitutes
a pair of two types: the system handle (unsigned int32) and the set of attribute types utilized in system calls
for creating objects such as: strings, int32 (for access flags), pointer and etc.

**Manipulation place** \( (P_{\text{manip}}) \) represents a particular operation (manipulation) on an object. Such a
place contains tokens representing the successful execution of a corresponding operation. A token comprises a handle of the manipulated object and critical parameters of the operation represented by the place. Thus, the color set of a Manipulation place consists of the space of system handles and a set of selected parameters of the operation associated with the place.

**Functional place** \((P_{\text{fun}})\) corresponds to a unique functionality. These places contain tokens that represent the successful recognition of a given functionality. Note that functionalities represent not a particular object, but a pattern of manipulations on several objects. The color set of Functional places includes only selected attributes of the necessary objects involved in the respective functionality as well as the objects’ operation parameters that individualize the functionality.

**Pseudo place** \((P_{\text{pseudo}})\) is associated with the pseudo states of the AD. A manipulation/object place represents executed object operation. An input transition of an object place must be attributed to the execution of one of the functionally equivalent APIs or system calls implementing the respective manipulation. An input transition of a functional place should be enabled when corresponding functionality is executed. Hence, the set of transitions consists of three disjoint sets:

\[
T = T_{\text{man}} \cup T_{\text{pseudo}} \cup T_{\text{fun}},
\]

where \(T_{\text{man}}\) - manipulation transitions representing system calls or a subsystem level operation (exported by API). \(T_{\text{pseudo}}\) - pseudo transitions that are utilized to reflect AD pseudo states. \(T_{\text{fun}}\) - functional transitions such that their input and output places constitute functionalities or functional object operations. We should point out that technically, a functional transition would coincide with the appropriate system call in a discrete event scale. However, the occurrence of such a transition is a semantically important event, thus we deliberately do not associate it with a system call.

Each manipulation transition \((T_{\text{man}})\) is enabled upon execution of any of the equivalent APIs performing the manipulation. Consequently, the guard expressions of such transitions must be defined over the object descriptor space (handle and buffer address) as well as over the manipulation parameter space. Guard expressions ensure that only manipulations with parameters determined in the corresponding AD would enable the transitions. The expressions of the output arcs may include variables of any type from the color set which covers the necessary attributes of the system calls and API functions. This provides enough flexibility to distinguish even similar, yet semantically different functionalities.

We developed procedure “ADtoCPN” that produces a CPN from the given functionality AD. Such a CPN would possess the necessary execution semantics to recognize the functionality. Due to the space limitations, we outline only high-level steps of the procedure.

---

**Procedure ADtoCPN**

Input: \(F\) – an AD of the functionality defined by the formalism (1).

Output: \(\text{CPN}\) – a CP-net that recognizes the given functionality \(F\).

1. Compose the CPN structure \((P, T, A)\) corresponding to the constructs of the AD of the functionality. Arcs of the AD are replaced by transitions and nodes are replaced by places.
   1.1 Form a set of places \(P\) and set of transitions \(T\) that correspond to the state and pseudo state nodes of the functionality \(F\).
   1.2 Form a set of the CPN arcs \((A)\) connecting the places and transitions created in the previous step (1.1)
   1.3 Form a set of functional places, transitions and corresponding arcs.
2. Define place colors \((C)\), guard expressions \((G)\) and arc expressions \((E)\) that define execution semantics of the functionality \(F\) in the given domain.
   2.1 Define guard expressions of the manipulation transitions that check the executed manipulation parameters against parameters specified in the functionality’s AD.
   2.2 Define guard expressions at the transitions that represent branching arcs of the AD decision nodes.
   2.3 Define a color function \((C)\) that would reflect variables of the functionality.
   2.4 Define arc expressions representing variable assignment in the functionality’s AD.
   2.5 Induce Color set \((S)\) and the rest of the arc expressions from the color function \((C)\) and the CP-net structure \((P,T,A)\)
3. Compile a CP-net \((\text{CPN}=(S,P,T,A,N,C,G,E,I))\) from the component sets obtained in steps 1 and 2.
Consider low-level CPNs recognizing subsystem object manipulations in the system call domain. These CPNs are obtained from system call level ADs specifying object manipulation. Here, manipulation transitions are enabled by system call execution, hence it is an open network driven by externals events (OS calls). Therefore, manipulation transitions do not have input arcs, representing inlet transitions. We also distinguish outlet transitions that represent handle/object elimination. For instance, the NtClose system call enables an outlet transition that destroys a token from the corresponding place.

Using procedure ADtoCPN, we obtained both high- and low-level CPNs for “Remote Shell” functionality (Fig 5). Figure 6 shows the high-level CPN; its places are shaped as ellipses and transitions as rectangles. The CPN nodes indexes correspond to AD nodes they recognize. For instance, transition #7.1 and place #7.2 recognize Node #7 of the Remote Shell AD. In Figure 6, the cloud shapes symbolize external CPNs such as Remote IPC CPN, Low level CPN, etc. These external CPNs recognize corresponding functional/subsystem manipulations and enable relevant transitions. For instance, transition #1.1 is enabled when functional object “Remote IPC” is created. The transition’s guard, “PID in PIDList”, checks whether process performing “Remote IPC” belongs to the list of descendant processes. This requires tracing generated process as specified in AD in Figure 5. The process tracing is performed by “Process Generation” CPN that provides descendant PID list as tokens to transition #7.1. The Object-places are highlighted with bold fonts. The place #5.2 (“RemoteShell”) is a recognition/functional place that represents successful functionality recognition.

Low-level CPN is shown in Figure 7; it recognizes the following subsystem level manipulations: “Create Socket” (socket API), “Bind”, “Listen” (listen API), “Accept” (accept API) which are exported by ws2_32.dll; and “Create Named Pipe” (CreateNamedPipe API), “Connect Pipe” (ConnectNamedPipe API) which are exported by kernel32.dll. In parentheses, we provided an API function that belongs to the group of equivalent subsystem APIs performing the associated manipulation. The CPN has three color sets (types): handle (H), which variables represent object handles; string (S) for the file names and uint32 (I) for access flags. Color and variable declarations are written in CPN markup language (CPN ML) syntax [6]. The CPN has 10 inlet transitions corresponding to system call execution. These transitions generate tokens representing attributes of the system calls that are processed by the CPN.
It could be seen that CPN’s structure is very similar to the structure of the AD. CPN is a very efficient recognition mechanism due to token dynamics. Hence, CPN causes low performance penalty for the anti-obfuscation generalization we introduced in Section 3. This ensures the high scalability of our approach allowing for addressing most of the known high-level obfuscation techniques.

5 Dynamic Information Flow Tracing

Depending on the specification, our IDS could employ a coarse-grained detector or fine-grained detector. Coarse-grained detectors trace only system call execution discarding information dependencies. Fine-grained detectors [3] trace information flows using dynamic data tracing techniques such as taint propagation [32, 34] thus potentially providing additional discriminative power. However, it was shown that purely dynamic techniques cannot trace data transmitted through covert channels such as implicit flows [28]. A particular attack on taint propagation technique [31, 32] was described in [27] using implicit flow technique, which is hard to defend against as confirmed in [33]. Since the implicit flow allows transmitting a bit by not executing a branch conditioned by a tainted value, control flow analysis techniques such as [32, 34] are useless in this case. Note that the static analysis of the non executed branch would not help either because, in general, the branched code could be encrypted. Forced execution of such a branch may fail if it has implicit jumps depending on tainted values.

In general, malware can evade data tracing by using an implicit flow that is easy to implement [27]. However, malware cannot avoid using system calls. Consequently, dynamic information flow tracing would not decrease false negatives compared to purely system call based detection. On the other hand, taint propagation may decrease false positives since there is no reason for legitimate software makers to use covert channels in their codes, unless they want to protect their products against reverse engineering.

The proposed AD formalism (2) allows for specifying functionality with informational dependency between the operation attributes. Recognition of such a functionality would require the utilization of taint propagation engine [32, 34] coupled with the system call monitor.

5.1 Taint Propagation Engine

While implementing the tainting engine, we generally followed the methodology given in [31-34], our implementation differs in the following aspects:

- Taint source and sink utilization
- Taint dependencies and propagation

**Taint source and sink**

In our system, the objective of the tainting engine is to trace information flow between object operations. In the AD formalism (2), an information flow could be specified through a variable $x$ referred by the content through a transformation $T(x)$. In this case, the source of the information flow is the operation whose output attribute defines the variable. The destination of the flow is the operation whose input attribute depends on the content of the variable. Note that in case when the variable representing the flow is referred to by the content in several operations/attributes, an information flow may have multiple destinations.

Functional object operations are based on subsystem operations that in turn are implemented through APIs and system calls. Hence, technically, information flow tracing is initiated by tainting the output argument of a system call implementing the source operation. At the same time, the flow is recognized by checking the taint of the input arguments of the system calls realizing the destination operations. To avoid false positives, we utilize a unique taint label for each particular instance of the specified information flow.

**Taint dependencies and propagation**

As described in [31-34], our system propagates the taint label according to three dependency vectors: explicit data flow, system call and control flow.

For the x86 architecture, data flow dependency could be represented by data transfer and stack instructions, such as MOV, MO VX, PUSH, POP etc., or arithmetic and logical instructions, such as ADD, SUB, AND, OR etc. In the case of direct data transfer, the engine propagates tainted bytes of the source to respective bytes of the destination. However, if the source is a register and is tainted, the engine marks all bytes of the destination. Note that the source operand could be indirect, for instance MO VX ecx, word ptr [ecx+eax*2]. In this case, the engine taints all bytes of the destination if either index or displacement registers of the source is tainted. Such a policy allows for tracing array manipulations indexed by tainted values.
Control data flow usually takes place when a variable is assigned within the scope of if, else or switch, case blocks that are conditioned by the tainted value. To resolve such a dependency, we followed [34] methodology that implies tainting every destination within the scope of the conditioned branches. However, according to our experience, sometimes one cannot take into account the entire scope, because if any of its branches leads to the return of the current function, the system taints everything in the rest of the function resulting in false positives. In such a case, we mitigate false taint propagation by pruning such branches from the flow graph and limiting the depth of the scope.

System call dependency is represented by data processing system calls/API, such that they do not perform any system related activity being only responsible for generating output data from the input data. Such data processing system calls are best exemplified by RTL functions: RtlInitAnsiString, RtlAnsiStringToUnicodeString. For instance, RtlAnsiStringToUnicodeString(outbuff, inbuff, …) creates null terminating UNICODE-string (outbuf) from input null-terminated ANSI-string (inbuff). Upon execution of this system call, our engine would respectively taint Unicode characters (words) of the output string (buffer) corresponding to tainted ANSI characters (bytes) of the input string (buffer). Note, here we perform one to one tainting to exclude false taint propagation. Some system calls may untaint the input argument. For instance, if RtlFreeHeap is to be invoked, then the input buffer is freed from the heap causing our system to untaint the content of the buffer.

Unlike [31-34], in our system, a particular taint label may become obsolete (retired). When the information flow is recognized, the taint label of the flow retires, meaning that the system will untaint any object tainted by the retired label. A label may also become obsolete if the system identifies that the source system call was executed in the frame of a wrong (non-source) operation. The latter situation may occur if two different operations begin with the same subsequence originated in the source system call and then split. In this case, the operations will be recognized only at the end of the execution, after the source system call, meaning that the engine must start tainting before recognizing the entire operation.

5.2 Taint utilization in CPN

While the taint engine is responsible for taint propagation, in order to recognize the information flow in the operation session, the CPN-based recognition engine is responsible for taint label management. Upon execution of a system call, the corresponding enabled transition creates a token representing the system call. If the system call is a source of a specified information flow, the transition should also signal taint engine to create a new taint label and add the label to the new token as a field. This way we transmit taint label of the particular instance of the information flow. When the destination system call is executed, its transition would also check if the taint label of the input token is equal to the taint label of the input attribute of the system call. If the labels match, meaning that the instance of the information flow reached its destination, the CPN recognizes the flow itself by enabling corresponding transition and firing a token to the recognition place.

Token dynamics play the critical role for the efficiency of the information flow recognition. Since taint label becomes a part of a token, the recognition mechanism verifies taint label only in two transitions corresponding respectively to the source system call and destination system call. In other transitions, that are not information flow endpoints, taint label is not verified at all. This separates the tainting engine from the recognition engine thus achieving the optimal overhead (complexity) distribution.

6 System implementation

6.1 AD designer

According to the architecture presented in Figure 1, an expert has to specify and supply activity diagrams of the functionalities defined in terms of the AD formalism (2). UML 2.x AD syntax provides enough constructs for specifying all components of the functionality formalism [35]. The state nodes are to be represented as UML actions, complex functional nodes as UML activities, object operation attributes and variable assignments as UML tag values. Additionally, UML syntax allows for using so-called stereotypes that are convenient for creating simple node profiles that define the set of tagged values. For each object operation, we use an individual stereotype that determines a set of attributes as tagged values. In our implementation, the choice of the UML AD designer is not critical as long as it is strictly compatible with the UML 2.0 standard. In our experiments, we utilized Visual Paradigm for UML, commercial software offering free community releases [24]. After finalizing the AD design, the expert should export AD to UML
XMI format used to exchange diagrams among UML compatible applications.

6.2 Specification Generalizer and CPN Constructor

We utilized Python language to implement prototypes of Specification Generalizer and CPN constructor modules (Fig. 1). The script for Specification Generalizer module constitutes 710 code lines that implement all three generalization algorithms and specific functions defined in Appendix D. We also developed a function that imports a formal AD from the input UML XMI file created by the UML AD designer such as Visual Paradigm. The importing is performed by interpreting and mapping UML constructs (e.g. tag values, actions, activities) to corresponding AD components (e.g. variables, object instances, operations) defined by formalism (2).

Prior to execution, the Specification Generalizer module imports the AD of the input functionality along with AD of generalization functionalities and functional operations pre-designed in Visual Paradigm software (see Appendices B and C). Then the module applies generalization algorithms to the input functionality and produces generalized AD. Finally, the module exports the generalized AD to XMI file. The resultant XMI file could be imported by the UML designer for on-demand editing or by CPN constructor for producing a CPN recognition model.

CPN Constructor module applies procedure ADtoCPN on the given functionality AD to produce recognition CPNs defined as tuple (3). Finally, the Constructor translates obtained Petri networks to CPN ML like format and exports it as XML file for CPN recognizer modules.

6.3 Functionality recognizer

We developed two realizations of the functionality recognition modules.

The first implementation was intended to evaluate the scalability and run-time efficiency of the methodology. The CPN recognizer was implemented in 3500 lines of native C++ code. Here, the CPN configuration is mostly hardcoded and its modification usually requires recompilation. To minimize the complexity of token matching we utilized self-balancing trees to store tokens in places. The trees are indexed by corresponding colors (usually handles) utilized in guard expression of the output transitions. To store tokens that represent file derivatives (copies) of the original target file we utilized chained hash tables indexed by file names.

For the sake of efficiency, in this realization, the system call monitor operated as a Windows device driver. In the driver, we utilized SSDT substitution technique to hook Windows system services [22, 25]. Due to the driver, such IDS is not completely transparent, however the IDS activity could be concealed through applying driver hiding techniques and covert user/kernel communications [25].

The second realization is less efficient, but much more generic. For this realization, we developed highly scalable and generic CPN simulator in C# .NET with Linq extension. The source code of most components of the CPN simulator is available at http://apimon.codeplex.com. The program solution includes several projects constituting 7900 code lines in total. The projects are responsible for system/API call hooking, call data parsing and transmission and CPN simulation. The CPN is built for simulator by translating arc and guard inscriptions to generative and filter expressions backed by Linq objects.

For the sake of performance and operability, we introduced some simplifications to the CPN simulator. The first simplification is that we treat CPN as an open model fed with tokens from external systems such as system call monitor and taint propagation engine. Secondly, we did not implement binding of variables belonging to different arc expressions in order to avoid computationally expensive cross list matching. Finally, we eliminated the possibility of specifying the number of tokens retrieved by an arc from a place.

In spite of the simplifications we preserved the most of the CPN execution semantics. Particularly, we treat arcs as token generators, and guards as token binders. Hence, CPN simulator is not limited to any particular execution domain and can process events of any nature delivered from multiple sources. Particularly, the CPN simulator can process system/API calls, API functions and functional object operations supplied from other CPN simulators. Such diversity allows us to build and simulate complex, hierarchical CPNs with low execution overhead.

6.4 Taint propagation engine

For the prototype, we did not attempt to achieve a low tainting overhead, but were interested primarily in
the evaluation of the recognition mechanism in tracing information flows specified in the functionality. The
taint propagation engine was implemented using IDA debugger with IDA python debug management script.
The engine runs the traced process in a debug mode and analyzes each instruction and its operands. For each
library function call, our system resolves the name of the function and input attribute. When a function is
called, IDA debugger breaks with dbg_step_into event and passes control to our script. The script verifies
function entry address and parses its attributes as well as disassembles the body of the function.

To determine entry point addresses of the system calls, our script verifies whether native system library
(“ntdll.dll”) was loaded by the process. To achieve this, every time a library is loaded, the debugger breaks
with dbg_library_load event, then our script parses ntdll.dll image and records its functions entry addresses
aligned to base address of the loaded module.

In our system, an expert has to provide declarations of system calls used as call dependencies as well as
declarations of related structures. The script parses standard declarations so that the expert may directly feed
the engine with declarations from MSDN website. Moreover, the expert has to provide dependencies
between input and output attributes of each particular system call being a call dependency. Such
dependencies are to be specified in a simple XML format.

7 Experimental evaluation

7.1 Experimental setup

Experimental evaluation of the described technology was conducted on the virtual network testbed
at Binghamton University [7], [8]. The testbed has been configured for a virtual network comprising a
dozens victim hosts represented by virtual machines with vulnerable versions of Windows OS and our
prototype IDS. Using the testbed, we experimented with various types of replication engines as well as
malware payloads constituting the following set of potentially malicious functionalities.

Replication engines:
• Self code injection – a malware infects an executable file through injecting its code into the executable
  body and replacing code entry points. It is used by file viruses.
• Self mailing - a malware emails its image as an attachment. It is used by e-mail worms
• Executable Download and Execute – Downloads a file from Internet and executes it. Used as a part of
  self-propagation engine of network worms [7], hence exposed by exploited processes and network bot
  agents such as Trojan-downloaders.
• Remote shell – Described in Section 2. Used as a part of propagation engine for network worms; also
  exposed by network bots.

Malicious payloads:
• Dll/thread injection - Injects DLL/thread to the address space of a process. Used for password stealing
  or process control highjacking.
• Self manage cmd script create and execute – A malicious process creates command script and executes
  it by running command interpreter. The command interpreter performs some operations on the malware
  image/dlls after its termination. This functionality relocates/deletes the malware image to conceal its
  footprint. Afterwards, a command script usually erases itself.
• Remote hook - sets a remote hook into victim process for a particular event; used for keylogging.
• Password stealing – steals credentials and sends them to the Internet. This functionality is discussed in
  the next section.

These functionalities were specified, generalized and translated to CPN. Since functionalities share the
same object operation sessions, to decrease simulation overhead we eliminated CPN structural redundancy
by integration of high level CPN into a single universal CPN having several functional places recognizing all
given malicious functionalities. The low-level CPN were also integrated into a single Petri network capable
of detecting object operations involved in the functionalities. The CPN configurations finally were loaded to
the Recognizer modules of the IDS.

In order to verify the detection rate, we experimented with the malware known, according to AV
descriptions, for perpetrating at least one of the malicious functionalities. The selected malware set included:
• File viruses – 7 instances (W32.Neo, Abigor, Crucio, Savior, Nother, Halen, HempHoper)
• **E-mail worms** – 9 instances (5 variants of w32.Netsky and 4 variants of w32.Beagle)
• **Network bots/Trojans** – SpyBot.gen, IRC.SdBot, RxBot families, Win32.Banker, Win32.lespy

We run each malware image in the corresponding environment enabling it to execute its payloads and/or replicate properly. The replication activity was exposed when victim hosts were attacked by various worms [7, 26]. The set of tested worms included modified strains to assure the propagation success, as well as non-modified strains to assure test fidelity. To invoke malicious payloads, we executed malware in certain preset conditions, for instance, we established an ftp/tftp server for **executable download and execute** functionality. In some cases, we had to enforce malware strains to run their payloads through debugging and run-time code modification.

In order to evaluate the false positive rate, we run multitude of benign software including web-browsers, messengers, email clients, file utilities, network and system utilities and office tools. We run the tested software under various conditions/inputs to expose their functionalities. We should point out that our experiments hardly covered all execution branches of the tested programs missing certain minor behavior patterns. Nevertheless, we believe that in our experiments the tested software have exposed the main activities.

### 7.2 Detection Results

Tables 5, 6 and 7 describe our experiments. The upper part of Table 5 presents detection results for the legitimate software: each cell indicates how many programs were detected based on the given functionality.

The lower part of Table 5 features results for malicious software. For each malware set, we indicate how many instances possessing the given functionality, were detected. For example, 4/4 means that there are four malware instances exposing the given functionality and all four were detected by our IDS.

The rest of the section discusses in detail results for the false positives and negatives.

#### 7.2.1 False positives

To assess the false positives, we performed two experiments.

In the first experiment, we manually run a diverse set of 210 legitimate programs including web browsers, e-mail clients, system tools, file managers, office tools, hooking software etc. We did not traverse all functionalities in all tested software focusing only on main features of each tested program.

Table 5 indicates that eight programs out of 210 showed false positives. Indeed, some known malicious functionalities could be exposed by certain legitimate software due to the following reasons.

- **Executable Download and Execute.** This functionality can be performed by the Internet browsers or file managers, mostly on behalf of the end-user. In addition, many programs perform periodic checking for updates, if there is an update available, the program downloads it and then executes. This activity can also be tagged as download and execute.
- **DLL/thread injection.** It can be performed by user/system monitoring software. Particularly, Easy Hook library injects DLL to trace API calls performed by arbitrary program. WinSpy program performs DLL injection in order to retrieve window objects data of a foreign program.
- **Self manage cmd script create and execute.** To uninstall hooks, the Easy Hook exiting functions run a cmd script that waits the hooking process to end, then removes the hooked DLLs.
- **Remote hook.** Hooking can be performed by chat programs to indentify whether a user is idle. These programs hook into other processes for the input events such as keystroke and mouse message.
- **Self-mailing.** When a user opens the “Save/Open file” dialog window, many programs represent every file found in the directory by the proper icon in the dialog window. In this case if a user browsed to the programs image location in the dialog window right before sending an e-mail with attachment, e-mail clients may show false positive. However, such behavior coupled with the sending email with client image is atypical user activity. Since such situation happened only in a particular scenario artificially performed during testing, we did not account it to false positive of the entire e-mail client program.
Table 5 clearly demonstrates the difference in discriminatory power of various functionalities that are frequently exposed by malware. According to Table 5, self-code inject, self-mailing and remote shell are never exposed by benign software, thus they have near perfect discriminatory power and can be used for malware detection. However, “Executable Download and Execute” (“ED&E”) is exposed by benign software such as web browser has low discriminatory power, hence it cannot be recommended for a signature-based detection. Regardless of discriminating power, our experiment demonstrates the ability to reliably detect individual functionalities. This ability could be beneficial for the detection of complex malicious payloads, such as password stealing, that may involve the combined use of several interrelated primitive functionalities.

The second experiment was performed utilizing a large set of system tools. The purpose of this experiment was to verify whether MS Windows package has programs that expose malicious functionalities in their main operational mode. This was achieved by running automatically all binary executables from Windows system folders (C: \ Windows\ and c: \ Windows\ System32\). For each program, our system performed the following steps:

1. Create suspended process
2. Initiate CPN simulator and system call monitor
3. Resume process and collect data until program finishes execution or after 20 seconds timeout
4. Write place reachability statistics to a report file
5. Clear Petri Net contents for the next run

In total, our system run 339 programs located in windows folders. All CPN statistics reports are summarized in Tables 6 and 7. Table 6 features the reachability statistics for low (system call) level CPN (see Figure 1). Low level CPN recognizes object operations exported by subsystem API. Hence, it has a recognition place for each necessary subsystem API exported by kernel32.dll and ws2_32.dll. In the table 6, the first column shows the names of the libraries which API are recognized by CPN. The second column shows name of the API functions that export object operations recognized by the CPN. The third column

<table>
<thead>
<tr>
<th>Legitimate software</th>
<th>Self-replication</th>
<th>Replication/ payloads</th>
<th>Payloads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Self code inject</td>
<td>Self mailing</td>
<td>Exec. Download &amp; Execute</td>
</tr>
<tr>
<td>Windows system tools, office apps, other utilities</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Web browsers (Opera, IE)</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>E-mail clients (Outlook Expr, Eudora)</td>
<td></td>
<td></td>
<td>2(?)</td>
</tr>
<tr>
<td>Instant messaging client (Yahoo messenger)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>File managers (FAR, Win Exp)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total detected</td>
<td></td>
<td></td>
<td>4/210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Malware</th>
<th>File viruses</th>
<th>Network worm shell codes</th>
<th>Network worm payloads</th>
<th>E-mail worms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7/7</td>
<td>2/2</td>
<td>8/8</td>
<td>9/9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5 Functionalities detection rate and false positive rate

<table>
<thead>
<tr>
<th></th>
<th>Self-replication</th>
<th>Replication/ payloads</th>
<th>Payloads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Self code inject</td>
<td>Self mailing</td>
<td>Exec. Download &amp; Execute</td>
</tr>
<tr>
<td>203</td>
<td>Windows system tools, office apps, other utilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Web browsers (Opera, IE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Instant messaging client (Yahoo messenger)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>File viruses</td>
<td>7/7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Network worm shell codes</td>
<td>2/2</td>
<td>8/8</td>
</tr>
<tr>
<td>6</td>
<td>Network worm payloads</td>
<td>4/4</td>
<td>1/1</td>
</tr>
<tr>
<td>9</td>
<td>E-mail worms</td>
<td>9/9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SpyBot.gen family</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>IRC.SdBot family</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>RxBot family (11)</td>
<td>all</td>
<td>all</td>
</tr>
</tbody>
</table>

False positive 0% 0% 1.92% 0% 0.48% 0.48% 0.96%
Detection rate 100% 100% 100% 100% 100% 100% 100%
shows number of programs that invoked particular API resulting in successful recognition of the corresponding API/operation. For instance, the table shows, that CPN recognized kernel32.WSASocket API in the system call flow of 8 out 339 programs.

It could be seen that reachability for places associated with system call execution is higher than the reachability of the API (object operation) places. It happens due to the fact that a single API may repeatedly invoke many system calls resulting in many tokens for each system calls. However, during the process of API recognition, most of the CPN transitions take several system call tokens and fire only one API related token. Note, since number of the processed tokens decreases towards recognition place, CPN simulation overhead also decreases while approaching to the moment of functionality recognition.

<table>
<thead>
<tr>
<th>Library</th>
<th>System calls/API</th>
<th># of programs reached the place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ntdll (System call)</td>
<td>ZwClose</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>ZwCreateFile</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>ZwOpenFile</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>ZwReadFile</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>ZwWriteFile</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>ZwCreateSection</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>ZwMapViewOfSection</td>
<td>151</td>
</tr>
<tr>
<td>Kernel32 (API)</td>
<td>CreateProcess</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>CreateNamedPipe</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>ConnectNamedPipe</td>
<td>0</td>
</tr>
<tr>
<td>WS2_32 (API)</td>
<td>WSAConnect</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Connect</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Bind</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Listen</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Accept</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Send</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object</th>
<th>Operation/functionality</th>
<th># of programs reached the place</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Map file</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Read file</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Write file</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Read itself of map itself</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Write to executable file</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Inject self-code</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Start process from edited or created executable</td>
<td>0</td>
</tr>
<tr>
<td>Named Pipe</td>
<td>Pipe created and connected</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Remote shell via named pipes</td>
<td>0</td>
</tr>
<tr>
<td>Socket</td>
<td>Socket connected</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Download and execute</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Socket bound</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Socket listening</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Accepted sockets</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Remote shell via socket</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SMTP protocol</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Self-mailing</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7 presents reachability statistics for high (subsystem) level CPN. Low level CPN provides high level CPN with tokens associated with system object operations that are involved in particular functionalities. In the table, the second column depicts the name of the operation or functionality that is represented by respective recognition place. Similar to Table 6, the third column of Table 7 shows number of programs that reached the particular place associated with the functionality or object operation.

Table 7 presents five functionalities of interest (marked by grey background): self code inject, self mailing, remote shell and executable download end execute. We should point out that each functionality is
represented by a recognition place in the high level CPN. Therefore, a functionality is detected in the program if a corresponding CPN place is reached by a program during the test.

Tables 6 and 7 indicate that most programs opened files, read/wrote some data, however only a few accessed files of interest such as executables or libraries. Several programs created socket and established a connection, however none of them utilized that socket for remote shell or to download an executable. As a result, CPN recognition places (shaded rows) have never been reached by any program out of 339 tested ones what indicates no false positives.

The goal of the second experiment was to verify windows system tools with standard inputs in standard operation mode. Our experiments showed zero false positives. It seems that the reason of zero false positives could be accounted to the fact windows tools have only necessary and limited capabilities strictly defined by the purpose of the tool, hence there is no reason for redundancy on the functional level. For instance, registry management tool would never download a file from a remote host, simply because this functionality could be achieved through other dedicated tool. Certainly, such tools would not perform unnecessary functionalities such as self code inject or self-mailing.

7.2.2 False negatives (detection rate).

As Table 5 indicates, for each malware containing the given functionality, our IDS successfully detected the functionality showing no false negatives. Such low false negative rate could be attributed to the signature generalization. For instance, the Beagle worm drops itself into the system folder, and then it e-mails its dropper. However, our prototype system successfully detected the self-mailing activity because it traced the dropper as object relocation functionality.

While creating AD specifications for the tested malware, we observed an interesting fact, that malware strains within the same family rarely demonstrate a conceptually novel realization. Instead, new malware strains frequently introduce minor alterations to their functionality realizations such as utilization of alternative APIs or changing Local IPC, i.e. switching from named pipe to shared files. We see two reasons for this trend. First, the attackers try to change malware system footprint in order to avoid certain AV signatures. Second, in the case of net bots, they simply try to enhance the performance of malware by optimizing or simplifying their implementation.

7.3 Case study - Password Stealing

Table 5 demonstrates that malwares, such as net bots, perform many malicious functionalities as their payloads. Such diversity could be conducive to detection. We may target not just one malicious functionality, but rather pattern of such functionalities that would determine the degree of hostility of the process.

<table>
<thead>
<tr>
<th>Process 1 (master process)</th>
<th>Process 2 (victim, hooked process)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Hook to victim process keystroke events</td>
<td>Establish Inter-Process Communication (IPC) with the victim process</td>
</tr>
<tr>
<td>2 Send the data to Internet</td>
<td></td>
</tr>
</tbody>
</table>

As indicated in Table 5, remote hook has several false positives. While remote keystroke hooking may not be malicious (at least with chat programs), keystroke stealing is certainly malicious. The fact that the hooked (victim) process transmits some data to the master process and then the master process sends something to the Internet, is much more suspicious. This rather complex functionality can only be detected by analyzing the combined activity of both processes (master and victim) and correlating their invoked manipulations. In this case, such activity combines functionalities 7, 5 and Local IPC.

The functionality mentioned above is known as Password stealing that is presented in Table 8. In the first step, the master malicious process sets keystroke hooks to the victim process. In the step 2, the hook handling function in victim process transmits keylog to the master process. Finally, in the step 3, the master process sends keylog data to the Internet. In the table, the step 2 represents the combined activity of both master and victim processes. While steps 1 and 3 constitute individual activity, e.g. the master process does not need cooperation from the victim process to perform a remote hook or to send data to the Internet. Due to space limits, we do not depict an AD diagram for this functionality, however, in terms of high level steps, the AD would be very similar to the representation in Table 8.
Figure 8 depicts a CPN that recognizes the password stealing functionality. The CPN designed with sheer functional objects recognized by external CPNs. The transition 1.1, 2.1 and 3.1 correspond to the activities in steps 1, 2 and 3 respectively in Table 8. These transitions are enabled upon recognition of the corresponding functional operations in the external CPNs such as: Remote Hook, Local IPC and Remote IPC. As shown in Figure 8, node 6 represents successful recognition of the functionality. Remote Hook CPN recognizes the Remote Hooking functionality which is step one in Table 8. This CPN has one recognition node – named “Hook”. Each token in this recognition place represents the successful execution of a remote hook. The color of such a token defines the following: ID of the process that performed the hook, ID of the thread that is hooked, and the type of the hook for instance keystroke hook (WH_KEYBOARD). This CPN recognizes several realizations of remote hooking such us: DLL injection, direct windows hook and windows message parsing.

Detection rate. We experimented with two families of malware that include four variants of Win32.Banker and two variants of Win32.lespy. According to their description in virustlist.com, these malware expose the functionality we recognize, i.e. password stealing with IPC. Our prototype IDS successfully detected password stealer functionality in all tested malware.

False positive rate. To estimate false positives we experimented with several popular programs: two messengers (QIP Infium, MS Messenger), two browsers (MS Internet Explorer, Opera), file manager (Far), email client (Outlook Express), automatic keyboard layout switcher (Punto Switcher). The results are summarized in Table 9 that presents place reachability of the CPN. Table 9 indicates that all tested programs performed “Hooking” functionality (place 1.2 was reached) and most of them opened Remote IPC (place 3.2 was reached) and sent some data. However, none of them connected to the process to which they hooked (place 4 was not reached). Hence, we did not observe false positives on this set of software.

The results demonstrate that it is more effective to detect more complex functionalities rather than primitive functionalities. This example shows the big advantage of utilizing CPNs for processes behavior recognition which is their ability to trace the activity of several processes in the context of a single CPN. Moreover, in the CPN, the necessary attributes propagate as token fields what allows for relating system calls by process and thread ID. This makes possible to recognize an interposed (system-wide) activity such as password stealing involving two processes (the master process and the victimized process with an injected DLL).

<table>
<thead>
<tr>
<th>Table 9 Place reachability of CPN for &quot;Password Stealer&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Far manager</td>
</tr>
<tr>
<td>Internet Explorer</td>
</tr>
<tr>
<td>QIP Infium</td>
</tr>
<tr>
<td>MS Outlook Express</td>
</tr>
<tr>
<td>MS Messenger</td>
</tr>
<tr>
<td>Opera</td>
</tr>
<tr>
<td>Punto Switcher</td>
</tr>
</tbody>
</table>

Figure 8. High level CP-subnet for the “Password stealer” functionality
8 Performance Overhead

Scalability of our IDS depends on two main factors: execution overhead of the monitored processes and overhead penalty of the CPN generalization. The first factor determines quantitative restriction of our IDS, i.e. how many process could be protected by our IDS. The second factor defines qualitative restriction, i.e. how much generic our IDS could be to address possible obfuscations.

Processes execution overhead is imposed mostly by system call monitor and little by CPN processing. System call monitor driver is always active in the Windows Kernel. When the system call of interest is invoked, the driver receives the execution control from system service dispatcher, reads input parameters of the system call, invokes the original system call, reads the output parameters of the system call and returns the execution control to the dispatcher. Such reading and saving attributes contributes the most to the execution overhead of the process that invoked the system call.

Each system call of interest, invoked by a process, gets processed by CPN. Hence, the more systems calls were invoked per time unit, the more overhead would be imposed by CPN processing. However, our CPN execution semantics appeared to be very efficient in processing a large number of system calls.

Periodically, the CPN recognizer requests system call data from the monitor driver. Such User/Kernel communication imposes additional overhead that is minimized due to buffering system call data and desynchronizing system call input and output attributes. In other words, the driver does not wait for system call execution and may send input parameters before receiving the output parameters.

The IDS was executed in Windows XP Professional SP2 running on an AMD Athlon 64 X2 (2200 Mhz) processor with 2 Gb of memory. In our performance tests, we evaluated overhead imposed by tracing generalization functionalities indented for behavioral de-obfuscation.

8.1 Run-time performance analysis

We measured overhead of system and application tasks using commercial benchmarks and manual setup. To achieve consistent results on Windows XP, we deactivated Windows prefetcher, scheduled tasks and accounted only for warm runs (to minimize cache influence). Some tests such, as file search and software installation were performed in virtual machine with reverting initial snapshot state for each run.

The test results are presented in Table 10 for Remote Shell functionality. For the sake of brevity, we showed here a selected set of standard tests that are representative with respect to execution overhead. The table depicts two system tasks and three application tasks. These tasks intensively utilized OS resources (services) resulting in a large number of invoked system calls. Some tasks involved user interaction with GUI of the corresponding application. In these cases, we utilized TestComplete software [29] to simulate user behavior.

We also run series of benchmarks using well-known PC Mark 05 suite [30]. Internet Explorer was tested with Peacekeeper benchmark [30]. We run each task/benchmark several dozen times with identical initial conditions and computed mean value and standard deviation of the execution time/score assuming normal distribution.

In order to estimate qualitative scalability of our IDS, we tested each task against two CPN configurations: Basic and Full. The Basic configuration covers alternative realizations of the functionality in question, but does not trace generic objects or obfuscations. In contrast, the Full configuration traces necessary functionality generalizations and addresses all three obfuscations. To estimate quantitative scalability, our IDS observed all active processes, but CPN recognizer in all performed tests.

For each task, Table 10 shows: base execution time when IDS is disabled (no system call monitoring or processing) and execution time when IDS is enabled with both Basic and Full CPNs recognizing Remote Shell functionality (with monitoring all active processes). One can see that even Full CPN IDS does not impose much overhead (less than 4% in average), while monitoring more than 50 (all active) processes. In fact, we also run IDS with highly loaded Windows XP with more than 100 processes without any significant overhead. This result shows sufficient scalability to protect all processes of a modern OS.

It could be seen that generalization and de-obfuscation does not impose much overhead penalty (0.31% in average). Note that in some tests Full and Base CPN overheads are considered to be invariant under statistical hypothesis with 80% power. This shows that our IDS is highly scalable and can address much more behavioral obfuscations.

While the tasks in Table 10 exposed some overhead, many other standard computationally expensive
tests did not show any execution overhead. For instance, Matlab did not show any overhead because its benchmarks involved mostly memory manipulations and math computations which hardly utilize any system services resulting in low number of invoked system calls. Similarly, MS Word search and replace task impose much overhead on CPU, but virtually none overhead on OS itself.

8.2 Stress test

The purpose of this test was to estimate the overhead of the IDS operating under possible stress attack. The stress attack could be conducted by a malware in order to congest IDS. Such an attack implies invoking many system call chains without closing handles making IDS processing all the objects and binding their handles. However, such an attack would be successful only if malware would congest our IDS before congesting OS to keep low execution profile. In case of congesting OS, such malware would expose itself and could be detected and terminated by any system administration tool.

We utilized Microsoft Performance Monitor (Perfmon) tool to measure run-time overhead of the IDS. In this test, we evaluated performance penalty for countering the obfuscation through object relocation. Particularly, we measured overhead imposed by handle and file tracing functionalities to be introduced by corresponding generalization algorithms. In the experiment, we run specifically developed tester program that opened 70 files (kernel32.CreateFile) in windows system folder and for each file it duplicates 20000 handles (kernel32.DuplicateHandle) and 20000 mappings (kernel32.CreateFileMapping). As a result, it

<table>
<thead>
<tr>
<th>Benchmark/Application (Task discretion)</th>
<th>Execution (seconds / score)</th>
<th>Overhead (%)</th>
<th>System call count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System tasks</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files Search (Search *.exe in c:)</td>
<td>58.96 ± 0.907</td>
<td>62.01 ± 1.04</td>
<td>63.66 ±2.04</td>
</tr>
<tr>
<td>Application Installation (Install DirectX 9.0c)</td>
<td>112.3</td>
<td>The same</td>
<td>113.6</td>
</tr>
<tr>
<td>MS Word (Save a big file as rtf)</td>
<td>35.9 ±0.787</td>
<td>The same</td>
<td>37.4 ±0.52</td>
</tr>
<tr>
<td>WinRar (Compress Windows system folder)</td>
<td>292</td>
<td>The same</td>
<td>298 (Full) 296 (Basic)</td>
</tr>
<tr>
<td>Internet Explorer 8 (Peacekeeper Browser Benchmark, <a href="http://www.futuremark.com">www.futuremark.com</a>)</td>
<td>702 (Score)</td>
<td>665 (Score)</td>
<td>657 (Score)</td>
</tr>
<tr>
<td><strong>Application tasks</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application loading (Mb/sec)</td>
<td>4.96 ± 0.0132</td>
<td>The same</td>
<td>4.87 ±0.355</td>
</tr>
<tr>
<td>Web page rendering (pages/sec)</td>
<td>2.0332 ± 0.04672</td>
<td>The same</td>
<td>1.8892 ±0.1088</td>
</tr>
<tr>
<td>File Encryption (Mb/sec)</td>
<td>36.827 ±0.134</td>
<td>The same</td>
<td>35.746 ±1.066</td>
</tr>
<tr>
<td>XP Startup (Mb/sec)</td>
<td>5.88 ± 0.022</td>
<td>The same</td>
<td>5.75 ±0.28</td>
</tr>
<tr>
<td><strong>Average execution overhead</strong></td>
<td>Basic CPN configuration (with multiple realizations)</td>
<td>3.67%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Full CPN configuration (with generalization and de-obfuscation)</td>
<td>3.98%</td>
<td></td>
</tr>
</tbody>
</table>

| **Table 10 Execution overhead due to IDS** |

8.2 Stress test

The purpose of this test was to estimate the overhead of the IDS operating under possible stress attack. The stress attack could be conducted by a malware in order to congest IDS. Such an attack implies invoking many system call chains without closing handles making IDS processing all the objects and binding their handles. However, such an attack would be successful only if malware would congest our IDS before congesting OS to keep low execution profile. In case of congesting OS, such malware would expose itself and could be detected and terminated by any system administration tool.

We utilized Microsoft Performance Monitor (Perfmon) tool to measure run-time overhead of the IDS. In this test, we evaluated performance penalty for countering the obfuscation through object relocation. Particularly, we measured overhead imposed by handle and file tracing functionalities to be introduced by corresponding generalization algorithms. In the experiment, we run specifically developed tester program that opened 70 files (kernel32.CreateFile) in windows system folder and for each file it duplicates 20000 handles (kernel32.DuplicateHandle) and 20000 mappings (kernel32.CreateFileMapping). As a result, it
creates 1400000 distinct file object handles and 14 00000 mapping (section) object handles.

Figure 9 shows CPU usage for tester program and our IDS module. It could be seen that tester consumes substantial amount of CPU cycles (around 90%), while IDS recognizer module impose less than 2% overhead in average to trace object relocation activity. Such drastic difference in overhead could be attributed to the fact that each object creation and handle allocation imposes certain overhead due parsing/updating internal Kernel structures, manipulating low level objects by Object Manager and pre-processing system call attributes in API implementation. Even for simple handle duplication, system call invocation requires user/kernel switching that is expensive for Windows OS. In contrast, CPN handle binding requires only user mode memory manipulations with highly efficient algorithms, e.g. balancing trees, or simple pointer resolution, e.g. hash tables. Hence, for each handle duplication or object creation, CPN imposes much less overhead.

![Figure 9 Handle Duplication Test](image)

9 Background and Related work

Our approach belongs to the class of dynamic, behavioral based IDS. Success of such IDS is determined by two aspects: expressiveness of the signature specification language and efficiency of the recognition mechanism. Moreover, usability of the IDS depends on clarity and degree of abstractness of the specification language. Below we survey existing behavioral specification languages and discuss advantages of our approach. Then we show in what way our system is different and better than related existing system call domain IDS.

9.1 State-Transition and CPN specifications

Kumar and Spafford developed IDIOT system that utilizes a variant of CPN, termed Colored Petri Automation (CPA), to detect UNIX system misuses [9]. Compared to CPN, CPA lacks concurrency, local condition variables and arc generative expressions. While, concurrency may be not critical for detection, condition variables and arc generative expressions are vital for structural simplicity of the CPN recognizing our AD specifications. Moreover, our ADtoCPN procedure represents AD variables in CPN arc expressions.

Helmer at al. utilize hierarchical CPN and Software Fault Trees (SFT) as behavioral specification for a distributed IDS [10]. First, CPN structure is obtained from SFT specifications. Next, an expert has to define CPN semantics (arc and guard expressions) that responsible for processing monitored system/network events. The spirit of their approach is close to ours that implies separation of specification and execution domains. However, the authors in [10] did not address generalization and behavioral obfuscation issues, what is one of our main contributions.
Ho at al. proposed Partial Order State Transition Analysis Technique (POSTAT) to specify local as well as distributed attack scenarios that are matched by CPNs [11]. The authors also give an insight for normalcy specification through POSTAT.

Eckmann at al. proposed STATL language for specifying misuse signatures in the domain of interest (host or network) [12]. In STATL transitions represent executed activity and states represent status of particular system objects. While STATL allows for specifying quite generic activity, the specifications must reflect also execution semantics making the signature description cluttered and complex. In contrast, we do not specify execution semantics in our AD specification, since ultimately the AD is translated to the CPN possessing execution semantics.

The problem of most state transition (ST) techniques is that the signature serves as a specification and a recognition mechanism at the same time. Hence the ST signatures are specified in the corresponding execution domain, e.g. system calls, network low level activity, shell commands etc. For instance, for host based detection, an expert has to specify how transitions should process system calls invoked by malicious activity. Consequently, the specification would be overloaded with implementation details making it hard to create and verify. In contrast, we distinguish specification domain and execution/detection domain. We specify functionalities at high level (object operations) and we detect specified functionality at low level (system calls). As a result, our specifications are quite readable and succinct reflecting only critical malicious activity at high/abstract level, while omitting low level implementation details. Particularly, in our system, an expert does not have to bother about how CPN processes monitored system calls to recognize the specified behavior, because CPN’s structure and semantics are automatically build from rather generic AD specification that could be defined in terms of abstract functional objects.

Summing up, our approach differs from the above methods in the following aspects.

First, the specifications in the above papers do not formalize behavior in the domain of our interest, i.e. system object manipulations, hence such specifications are not directly suitable for automatic processing. In contrast, our AD formalism is defined in the object operation domain, thus allowing for automatic processing and generalization.

Second, the authors utilize CPN as behavioral specification language, however we utilize CPN solely as a recognition mechanism. In [7] we discovered that functionality specification through CPN in system call domain is tedious and hardly feasible for complex cases, hence we developed simple and generic AD formalism for functionality specification. Afterwards, our ADtoCPN procedure translates generic AD to system call domain CPN serving as a recognition model.

Finally, our AD based specification is generic due to functional blocks that abstract certain system implementation details yet provide enough agility for fine tuning of the specifications.

9.2 Declarative and Algebraic specification languages

Examples of declarative languages are LAMBDA [13], ADeLe [14] and Sutekh [15]. The specifications based on such languages define a pattern of actions involved in an attack, system pre and post conditions as a set of high level predicates and mapping between actions and observable system events manifested by the attack actions. While such language can express quite generic attack scenario, the complex scenarios description would require specifying many abstract predicates that apparently have to be identified by the detection mechanism. Run-time detection of such predicates may impose high overhead, post alarm verification of the predicates could be unfeasible due to changes in the system state [13]. In contrast, we do not specify pre-conditions, because validity of the system patterns is verified by CPN at run-time through checking system call outputs.

Aforementioned declarative languages allows for relating attacks to each other through matching the post-condition of an attack with the pre-condition of another. Our approach also allows for inclusion other AD in as functional object to the specified AD. In the level of CPN this would be realized as hierarchal CPNs.

The language of the system CARDS [16] introduces an abstract system view that allows for defining distributed hierarchical attack specifications in the domain of abstract events. The detection language SHEDEL [17] takes advantage on both algebraic and state-transition approaches. As with [16], SHEDEL specifications are explicitly defined at the abstract event level.

Declarative languages are advantageous over state-transition specifications in the ability of describing signatures at the pure abstract level omitting operational details related to the domain of attack execution. To
overcome this inherited limitation of ST languages, we proposed so-called functional objects that serve as a system abstract level on which AD signatures could be specified.

Unlike state-transition specifications, most declarative and algebraic specifications do not possess execution semantics. For the system call domain, IDS based on declarative specifications require recognition mechanism that would explicitly match system calls with signatures. As the result IDS scalability ultimately depends on efficiency of the recognition/matching algorithm. Usually run-time pattern matching is performed through some sort of state machine or rule based detector as in [15]. However, ST models are more efficient for handling multiple instances of the same pattern. If a pattern is observed more than once and each instance is yet incomplete, a state machine has to accommodate extra states for each instance of the pattern to trace them in parallel. In contrast, a ST model represents an executed event pattern as one token residing in the corresponding place that allows for processing of multiple pattern instances with low overhead. Moreover, in the CPN, the necessary attributes propagate as token fields that allows for relating system calls by process and thread ID. This makes possible to recognize an inter-process activity. Consequently, ST signatures are executable in the event domain, hence with ST the detection becomes the execution what should be faster than signature-event matching. Based on the above arguments, we believe that state-transition specifications are more preferable to declarative languages with respect scalability and efficiency.

Our AD specification is directly convertible to recognition mechanism configuration, i.e. high level CPN. As a result, in our case, the recognition mechanism (CPN) incorporates the signature in its execution semantics what ensures low detection overhead. In other words, the detection becomes the direct execution of CPN, hence no cross-matching for signature vs. event is performed.

9.3 System call domain specifications

System calls present a perfect domain for behavioral based misuse detection, because system calls are executed in safe Kernel mode and their monitoring is quite resilient against purely user mode malware.

Publications [18]-[21] propose tracing sequences of system calls to reveal misuse in OS objects manipulations indicative of malicious activity. These methods discard semantic relationships between object manipulations. Few attempts to deduce semantic information on the level of primitive functional blocks [18], [19] failed to define a complete picture of the process behavior. PC Tools ThreatFire antivirus [20] allows a user to specify rules that describe only individual operations on objects such as process, file etc., facilitating the detection of primitive and obvious misuse such as access to system file, or starting a particular executive. In contrast, our approach allows for recognizing complex functionalities (such as self-mailing) that involve interrelated sessions of object operations.

Papers [2], [3], [22], [23] target dynamic behavior analysis. The methodology presented in [23] allows for detecting virus activity through tracing system call events with emphasis on the order of events without functional context. In contrast, our approach allows for specifying rather patterns of interrelated manipulations and primitive functionalities. Dynamic Code Analyzer (DCA) approach allows for constructing so called “gene of self-replication” from primitive object operations and activity blocks but lacks an efficient recognition mechanism [22].

Authors in [3] utilize behavior graphs that in fact is an extension of malware specification graphs presented in [2]. Our work is different/advantageous over [2] and [3] in the following. Firstly, according to [2], the specification matching is performed through static analysis that implies mapping to some sort of call flow graph of the tested executables. Obviously, such behavior graphs do not address execution semantics, and could not be utilized as a run-time recognition mechanism. To recognize the specified behavior in [3], the authors utilize a behavior matching algorithm, unfortunately they do not formalize the algorithm itself. In particular, it is unclear how they might handle multiple system call chains. In contrast, as mentioned above, CPN efficiently handles multiple chains due token dynamics. Secondly, specification in [3] can define alternative realizations, but their behavior graphs are constrained to a single process. In contrast, our AD allows for correlating operations invoked by different processes, thus we can specify malicious inter-process and inter-host activity. Finally, the authors in [3] did not analyze and address possible behavioral obfuscations. At the same time they did not propose solid behavior formalism, hence it not feasible to automatically process the specification. Hence in [3] an expert is responsible for generalization of the specifications for instance to avoid handle duplication what could significantly complicate designing process. In contrast, we formalized functionality in the object operation domain. Such formalization allowed for automatic processing AD specifications to address possible behavioral obfuscations. We understand that
we addressed rather limited set of obfuscations, but given the flexibility and fidelity of our functionality formalization, developing new generalization algorithms for anti-obfuscation should be feasible for an expert.

10 Conclusion

In this paper, we address present and future limitations of the current Behavior Based IDS (BBIDS) associated with signature expressiveness, behavioral obfuscation and detection efficiency. We advocate for the separation of the specification and detection domains. We presented a new approach for formal specification of the malicious functionalities based on activity diagrams defined in an abstract domain. We introduced so-called abstract functional objects that along with system objects could be used for creating generic specifications that would cover multiple functionality realizations yet preserving perfect discriminatory power. We developed and tested an automated procedure enabling human experts responsible for the formulation of malicious behavioral pattern to concentrate on conceptual realizations omitting certain implementation details.

We analyzed and classified possible behavioral obfuscation techniques, both inter-process and intra-process, that can compromise existing BBIDS. As a mitigating solution, the concept of specification generalization that implies augmenting (generalizing) otherwise obfuscation prone specification into more generic obfuscation resilient specification was suggested. Generalization algorithms making AD immune to the obfuscations are developed.

We proposed a methodology utilizing CPN for recognizing functionalities at the system call level. Moreover, an approach for incorporation of information flows into CPN to achieve fine-grained recognition has been developed. Finally, we proposed an automatic procedure converting a given AD into a CPN that recognizes the defined functionality in the system call domain enriched with information flow data.

Every aspect of the proposed technology was implemented in a prototype IDS. We evaluated the IDS on dozens of malware and hundreds of legitimate programs. The experimental results indicate extremely low false positives and negatives. Finally, we performed series of experiments to estimate run-time overhead due to IDS. The results indicated two practical advantages. First, IDS causes low overhead which less than 4%. Second, the overhead increase due to the anti-obfuscation generalization constitutes only 0.3%. Such low overhead difference between generalized and original CPN indicates that an expert can always address even more obfuscation techniques with negligible execution cost.

11 Future research

As a future research, we plan to expand the list of possible behavioral obfuscation techniques and address them in AD generalization. We intend to explore the increasing role of behavioral metamorphism, that implies dynamic scattering of malicious functionalities among different benign processes so that none of the processes would have a consistent system call pattern, in offensive information warfare. We are interested in dynamic AD construction from the monitored behavior of processes of interest. First, this would allow for automatic retrieval of the functionalities for a particular program. Second, by comparison of AD obtained for legitimate software and malware, one can potentially extract so-called the “gene of maliciousness”, i.e. a set of functionalities present only in malware.

12 Acknowledgment

This research has been funded by the Air Force Office of Scientific Research (AFOSR). The authors are grateful to Dr. Robert L. Herklotz of the AFOSR, Information Operations and Security, for his on-going support of this effort.

13 References


[33] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda “Scalable, Behavior-Based Malware Clustering” in Proc. NDSS, 2009


Appendix A

Firstly, let us introduce some basic notations we use in the formalization:

- **O**: set of OS objects.
- **M**: set of object manipulations (operations).

Object or manipulations have attributes

- \( \text{AttList}: O \cup M \rightarrow U \), such that
  \( \forall x \in O: \text{AttList}(x) - \) set of attributes of the object \( x \).
  \( \forall y \in M: \text{AttList}(y) - \) set of parameters of manipulation \( y \).

- \( \text{AttSpace}: O \times \text{AttList}(O) \cup M \times \text{AttList}(M) \rightarrow U^4 \), such that
  \( \forall x \in O, \forall att \in \text{AttList}(x): \text{AttSpace}(x, att) - \) set of all possible values (space) of an attribute \( att \) of an object \( x \).
  \( \forall y \in M, \forall att \in \text{AttList}(y): \text{AttSpace}(y, att) - \) set of all possible values (space) of a parameter \( att \) of a manipulation \( y \).

The set of objects \( O \) includes both subsystem level objects (e.g. “File mapping”, “Socket”) and Kernel objects exported to the user mode (e.g. “File”, “Process”, “Find file”). The set of object manipulations \( M \) is induced by API functions as well as system calls performing the manipulations. The object manipulation parameter set is generated by attributes of semantically equivalent API functions that export the particular manipulation. The function \( \text{AttList}(x) \) returns list (set) of parameters of the operation \( x \).

Based on the above terms, functionality is defined as an Activity Diagram (AD) in the following form:

\[
F = (\text{Nodes}, \text{Arcs}, \text{Assign}, \text{Vars}) \tag{1}
\]

where,

- \( \text{Vars} - \) a set of local variables used in the object manipulations.
- \( \text{Nodes} = \{\text{IndState}\} \cup \{\text{Pseudo initial, final}\} \) is a set of AD nodes such that,
- \( \text{State} = \text{Instances} \cup \text{Manipulations} \) is a set of \( \text{State} \) nodes, where
  \( \text{Instances} \) is a multi-set of object instances defined as:
  \[
  \text{Instances} = \{(Ob, Attr)[Ob \in O]\}, \quad \text{where}
  \[
  \text{Attr} = \left\{\begin{array}{l}
  \{\text{Name}_i, \text{Value}_i\} \mid i \in 1..k, [\text{Name}_i, \in \text{AttList}(Ob)] \\
  \text{Value}_i \subset \{\text{AttSpace}(Ob, \text{Name}_i) \cup \text{Vars}\} \cup \text{Value}_i = T(x), x \in \text{Vars}
  \end{array}\right\},
  \]

where \( k \) is the number of critical attributes, \( T() \) abstract transformation of the input variable.

In the set \( \text{Instances} \) each element represents a particular object instance which is created in the context of the functionality execution. An object instance consists of the object name \( (Ob) \) and a set of attributes \( (Attr) \). Each object \( i \)-th attribute \( Attr_i \) is represented by a tuple \( (\text{Name}_i, \text{Value}_i) \). The first element of the tuple represents the name of the attribute that is unique for a particular object. The second element could define the following: value set from attribute domain, local variable or transformation of the local variable. Transformation \( T() \) is utilized for specifying informational dependency (flow) between attributes of the operations. Such transformation should not be defined to specify any information flow, e.g. data dependency of any nature including control related flows.

The variables are assigned during functionality execution. The set of attributes contains only those attributes that are critical for functionality execution. For example, in the functionality presented in Table 1, the instance of the “Process” object (created by CreateProcess) can be specified as:

\[
\text{Process}, \left\{\begin{array}{l}
(\text{bInheritHandles, TRUE}), (\text{STARTUPINFO.dwFlags, STARTF_USESTDHAND}) \\
(\text{STARTUPINFO.hStdInput, s}), (\text{STARTUPINFO.hStdOutput, s})
\end{array}\right\}
\]

- **Manipulations** - the set of invoked manipulations that is defined as:

\(^4\) \( U \) – universal set (set of all sets)
\[ \text{Manipulations} = \{ (M, \text{Params}) \mid M \in M \}, \text{ where} \]

\[ \text{Params} = \{ \langle \text{Name}_i, \text{Value}_i \rangle \mid [i \in 1..k], \langle \text{Name}_i \in \text{AttList}(M) \rangle, \text{Value}_i \subseteq \{ \text{AttSpace}(M, \text{Name}_i) \cup \text{Vars} \} \lor b_i = T(x), x \in \text{Vars} \}, \]

where \( k \) is the number of critical parameters, \( T() \) abstract transformation of the input variable.

In the set \( \text{Manipulations} \) each element represents an object manipulation invoked by the functionality. A manipulation is defined by the operation name and the set of input parameters. Every parameter is represented by a parameter name and a parameter value set that is a subset of the corresponding parameter set or that could be specified as a local variable or its transformation. The set of parameters comprises only those critical parameters that determine functionality.

\( \text{Ind} \) – a set of process identities, such that each element of this set represents a local ID of the process that performs the object operation. Hence, every distinct process involved in the functionality has its unique index form the set \( \text{Ind} \). This addresses the third requirement of the specification allowing for specifying an inter-process functionality.

\( \text{Pseudo} \) - pseudo nodes that route the control flow, presented by: decision, merge, fork or join.

\[ \text{Pseudo} = \{ \text{x} \mid \text{Type}(x) \in \{ \text{decision}, \text{merge}, \text{fork}, \text{join} \} \}, \text{ where Type}(x) \text{ is the type of the node } x. \]

\( \text{Arcs} = \text{ControlFlow} \cup \text{HandleFlow} \) is a set of directed arcs connecting operation nodes as a union of mutually exclusive sets \( \text{ControlFlow} \) and \( \text{HandleFlow} \).

\( \text{HandleFlow} \subseteq \text{Nodes} \times \text{Nodes} \) is the set of arcs (handle arcs) that correspond to execution flow with handle inheritance. A handle arc indicates that the destination operation (node) utilizes the same object instance handle as a source operation and is executed right after the source. In other words, the source and destination operations are performed on the same object instance and are involved in the same manipulation session. In terms of the UML 2.x activity diagrams syntax [14], such arcs could be viewed as a fusion of the object flow with the control flow.

\( \text{ControlFlow} \subseteq \text{Nodes} \times \text{Nodes} \) is a set of directed arcs that define the control flow without handle inheritance. The arc from this set indicates that the destination operation is executed right after the source operation. Note, such arc simply shows the execution order and does not indicate any data binding (via handle or attribute).

\( \text{Assign} : \text{Arcs} \rightarrow \text{Expression} \cup \emptyset \) is a variable assignment and guard function such that,

\[ \text{Assign}(a) = \begin{cases} \text{Assignment expression, } \text{Source}(a) \in \text{State} \\ \text{Guard expression, } \text{Source}(a) \in \text{Pseudo} \end{cases}, \forall a \in \text{Arcs} : \text{Source}(a) - \text{source of the arc } a. \]

Assignment expression = \( \{ "v := \text{out}" \mid v \in \text{Vars}, out \in \text{OutPar}(\text{Source}(a)) \} \), where \( \text{OutPar}(x) \) is a list of output parameters of object operation \( x \).

This function defines a variable assignment expression for corresponding arcs having the \( \text{State} \) node as a source. The assignment expression utilizes output parameters of the arc’s source operations to assign required local variables. Such parameters may include object descriptors (handle, memory offset, etc) of the source operation. If the source of the arc is a Pseudo state node, this function determines a guard expression as defined in the original UML 2.0 activity diagrams. Note that the \( \text{Assign} \) function does not define an expression for every arc, but for those where it is necessary.
Appendix B Remote IPC ADs

RemoteIPC – “Create” operation

RemoteIPC – “Wait” operation

RemoteIPC – “Recv” operation

RemoteIPC – “Send” operation
Appendix C Generalization functionalities AD

Handle Duplication Functionality

Process Generation Functionality

Code Injection Functionality
Appendix D Functions utilized in generalization algorithms

AddParallelFunct(
F.AD OriginAD
F.AD NewAD
OriginAD.Nodes Fork
OriginAD.Nodes Join)
It adds AD of NewAD functionality to AD of OriginAD functionality as a parallel flow that starts right after the node Fork and joins to Origin AD just before the node Join. F.AD means set of AD of all functionalities.

NewNode=AddParallelNode(
F.AD OriginAD
O ∪ M NewOperation
OriginAD.Nodes Fork
OriginAD.Nodes Join)
Creates node representing an input operation (NewOperation) and adds it to OriginAD functionality as a parallel flow that starts right after the node Fork and joins to Origin AD just before the node Join. This function returns added node NewNode. O ∪ M is a set of objects and manipulations.

NewNode=AddNextNode(
F.AD OriginAD
O ∪ M NewOperation
OriginAD.Nodes ParentNode)
Creates node representing an input operation (NewOperation) and adds it to OriginAD functionality right after the node ParentNode.

AttValue=GetAttributeValue(
AD.Node.State Node
AttList(Node) Attr)
Returns value of the attribute Attr of the node Node.

SetAttributeValueExpression(
AD.Node.State Node
AttList(Node) Attr
String Expression)
Sets attribute expression for Attr attribute of the state node Node of the current AD.

SetNodePIDExpression(
AD.Node.State Node
String Expression)
Sets an expression assigning PID of the node Node.

NewVarName=CreateNewVar(
InputAD.Arcs Arc
String Expression)
Introduces a new variable to the current input AD arc (Arc) that is defined with the assignment expression (Expression). The assignment expression may use output attributes of the parent of the arc and other global variables. This function returns name of the newly created variable. By current AD we mean AD being input of the algorithm.

InputAD.Nodes.State Node=GetAssignNode(
InputAD.Vars Var)
Searches for and returns the node in current AD which output arc assigns variable Var.