Mining Specifications of Malicious Behavior

Mihai Christodorescu*
University of Wisconsin
mihai@cs.wisc.edu

Somesh Jha*
University of Wisconsin
jha@cs.wisc.edu

Christopher Kruegel†
Technical University Vienna
chris@auto.tuwien.ac.at

ABSTRACT

Malware detectors require a specification of malicious behavior. Typically, these specifications are manually constructed by investigating known malware. We present an automatic technique to overcome this laborious manual process. Our technique derives such a specification by comparing the execution behavior of a known malware against the execution behaviors of a set of benign programs. In other words, we mine the malicious behavior present in a known malware that is not present in a set of benign programs. The output of our algorithm can be used by malware detectors to detect malware variants. Since our algorithm provides a succinct description of malicious behavior present in a malware, it can also be used by security analysts for understanding the malware. We have implemented a prototype based on our algorithm and tested it on several malware programs. Experimental results obtained from our prototype indicate that our algorithm is effective in extracting malicious behaviors that can be used to detect malware variants.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Experimentation, Languages, Measurement, Security

Keywords

behavior-based detection, differential analysis, malspec

*The work of Mihai Christodorescu and Somesh Jha was supported by the National Science Foundation under grant CNS-0627501.

†Christopher Kruegel was supported by the Austrian Science Foundation (FWF) under grant P18157, the FIT-IT project Pathfinder, and the Secure Business Austria competence center.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

1. INTRODUCTION

Malicious software (malware) is code that achieves the harmful intent of an attacker. Typical examples include viruses, worms, trojans, and spyware. Although the history of malware reaches back more than two decades, the advent of large-scale computer worm epidemics and waves of email viruses have elevated the problem to a major security threat. Recently, this threat has also acquired an economic dimension as attackers benefit financially from compromised machines (e.g., by selling hosts as email relays to spammers).

Historically, detection tools such as virus scanners have performed poorly, particularly when facing previously unknown malware programs or novel variants of existing ones. The fundamental cause is the disconnect between the malware specification used for detection and the actual malware behavior (which is the attacker's goal). Certain malicious behavior desired by an attacker (e.g., virus self replication through mass-mailing) can be realized in many different ways. However, current detectors focus only on the specific characteristics of individual malware instances, e.g., on the presence of particular instruction sequences. Therefore, they fail to detect different manifestations of the same malicious behavior. Attackers are quick to exploit this weakness by using program obfuscation techniques such as polymorphism and metamorphism [19, 22, 25, 29]. Recent research results have highlighted how shortcomings in both network-based and host-based detection techniques can be effectively exploited by attackers to evade detection [6, 14].

Advanced detection techniques such as semantics-aware malware detection [7, 16] and malicious-code model checking [12] counter the obfuscation techniques of attackers by using higher-level specifications of malicious behavior. Instead of focusing on individual characteristics of particular malware instances, these detectors specify general behavior exhibited by an entire family of malicious code. Examples of such specifications include self-unpacking and self-propagation via email. The power of these approaches resides in the use of high-level specifications that abstract details specific to a malware instance. Thus, obfuscation transformations, which preserve the behavior of the malware but may change its form, are no longer effective techniques for evading detection.

Unfortunately, the high-level specifications of malicious behavior used by these advanced malware detectors are currently manually developed. Creating specifications manually is a time-consuming task that requires expert knowledge, which reduces the appeal and deployment of these new detection techniques. To address this limitation, this paper

introduces a technique to automatically derive specifications of malicious behavior from a given malware sample. Such a specification can then be used by a malware detector, allowing for the creation of an end-to-end tool-chain to update malware detectors when a new malware appears. Since our technique provides a succinct description of the malicious intent of a malware, it can also be used by security analysts for malware understanding.

We cast malicious-specification mining as the problem of finding differences between a malware sample and a set of benign programs. This approach supports the requirement that the specification of malicious behavior must capture aspects that are specific to the malware and absent from any benign programs. The software-engineering research community has put a lot of effort in analyzing commonalities and differences between programs and many techniques for clone detection [3, 11, 13, 30] and program differencing [2, 9, 10, 17] have been proposed. The differencing techniques are closest to our goal of mining specifications of malicious behavior, but they fall short because they generally require access to source code and because they produce differences between low-level elements of the program (e.g., individual statements) or between structural elements (e.g., type hierarchy, procedures). Since we do not have the malware source code and since the malware writer controls the structure of the program, mining malicious specifications requires a new approach.

Our mining technique takes into account an adversarial setting in which the malware writer tries to make his software hard to analyze and detect. We define a new graph representation of program behavior and a mining algorithm that constructs a malicious specification. The representation explicitly captures the system calls made by the program and summarizes all other program code, because system calls are the primary interaction with the operating system. Our algorithm infers the system-call graphs from execution traces, then derives a specification by computing the minimal differences between the system-call graphs of malicious and benign programs.

This paper makes the following contributions:

- A language for specifying malicious behavior in terms of dependences between system calls (Section 3).
- An algorithm, called MiniMal[1] that mines specifications of malicious behavior from dependence graphs (Section 4).
- An experimental evaluation that shows that specifications extracted MiniMal are qualitatively equivalent to those manually developed by Symantec's expert virus analysts and can be used with a malware detector to identify subsequent malware variants (Section 5).

2. OVERVIEW

The goal of our specification language is to describe malicious behavior. In general, the behavior of a program can be described as its observable effect on the execution environment. Depending on what is considered the program's environment, behavior can be specified at different levels of granularity.

One approach is to define behavior as the effect of a sequence of instructions on the state of a process. For this purpose, the state of a process is defined as the content of the memory address space. The effect on this state are modifications to the contents of the address space. Another approach to specify program behavior is to view the running program as a black-box and only focus on its interaction with the operating system. In this case, a particularly convenient interface to monitor is the set of operating system calls that this process invokes. Every action that involves communication with the process' environment (e.g., accessing the file system, sending a packet over the network, or launching another program) requires the process to make use of an appropriate operating system service.

[1]MiniMal is a technique for mining *mini*mally *mal*icious behavior.

```
1  push  ebp
2  mov   ebp, esp
3  add   esp, 0FFFFFFFCh
4  push  offset aSoftwareDatetime
5  push  80000001h
6  call  RegDeleteKeyA
7  lea   eax, [ebp+hKey]
8  push  eax
9  push  offset SubKey
10 push  80000001h
11 call  RegCreateKeyA
12 push  offset ValueName
13 push  [ebp+hKey]
14 call  RegDeleteValueA
15 push  [ebp+hKey]
16 call  RegCloseKey
```

Figure 1: Bagle.J code fragment performing registry cleanup, one of its "administrative" tasks.

For this work, we choose the second approach and use system calls as the basic building blocks of our specification language. We only consider the actions of the program at the operating system interface, thus making our technique robust to any code-obfuscation techniques employed by malware authors. The end result is a specification of malicious behavior that captures the intent of the malware writer in terms of its effect on the host operating system. Such a specification can then be used to detect the original malware as well as obfuscated variants that exhibit the same system-call footprint.

At a high level, the mining algorithm works in three steps:

Algorithm MiniMal

1. Collect execution traces from malware and benign programs. (Section 4, Step 1, and Section 5)
2. Construct the corresponding dependence graphs. (Section 4, Step 2)
3. Compute specification of malicious behavior as difference of dependence graphs. (Section 4, Step 3)

We will walk through the application of the mining algorithm on the email worm Bagle.J, the tenth variant in a long line of malware known as the *Bagle malware family*. According to the Symantec Security Response [24], Bagle.J exhibits the following high-level functionality:

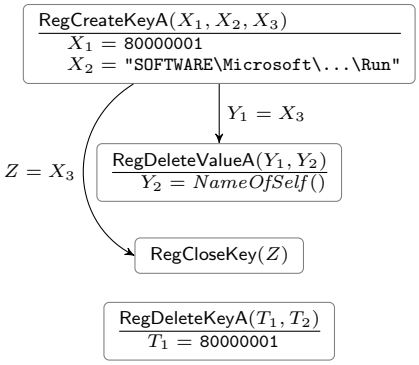
- It spreads through email, sending copies of itself using a built-in SMTP engine.
- It opens a backdoor on the infected system, allowing the attacker to access and control the victim system remotely.
- It self-propagates through several file-sharing networks (Kazaa and iMesh).
- It performs multiple "administrative" tasks that include

```
1  RegDeleteKeyA( 0x80000001,"SOFTWARE\DateTime" ) ==
       0
2  RegCreateKeyA( 0x80000001, "SOFTWARE\Microsoft\...\
       Run", 0x0007FFC4 ) == 0
3  RegDeleteValueA( 0x0007FFC4, "ssate.exe" ) == 0
4  RegCloseKey( 0x0007FFC4 ) == 0
```

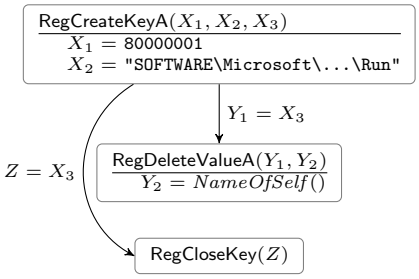
(a) Execution trace corresponding to Figure 1.

$$
\begin{aligned}
n_1 &: T_1 = \texttt{80000001} \\
n_2 &: X_1 = \texttt{80000001} \\
n_2 &: X_2 = \texttt{"SOFTWARE\Microsoft\...\Run"} \\
n_2 \rightarrow n_3 &: X_3 = Y_1 \\
n_2 \rightarrow n_4 &: X_3 = Z \\
n_3 &: Y_2 = \mathit{NameOfSelf}()
\end{aligned}
$$

(b) List of dependences.



(c) Dependence graph.



(d) Specification.

Figure 2: Mining a specification of malicious behavior starts from an execution trace (a) and produces a subgraph of dependences (d).

installing itself to run every time the system boots, terminating processes of known anti-virus and security software, and announcing the IP of the victim system by posting it on several web sites.

In Figure 1, a code fragment of the Bagle.J worm modifies several Windows registry keys to remove traces of itself. The code is interesting from a security perspective in that it both unregisters the worm from execution at the next system boot and it deletes a flag marking the presence of infection. Such operations often appear before the worm code "upgrades" itself to a new version. The calls to RegDeleteKeyA, RegCreateKeyA, RegDeleteValueA, and RegCloseKey (lines 6, 11, 14, and 16, respectively) describe the interesting and relevant interactions between the malware and the operating system (although the calls in Figure 1 are to library functions, they map directly to corresponding system calls). A specification for this code fragment must include these calls. The rest of the code in Figure 1 sets up the call arguments and passes values from one call to the next. Such code does not directly affect the operating system and can be obfuscated in many ways. For example, the hexadecimal constant 80000001 on line 10 (the first argument to RegCreateKeyA) could be computed by evaluating various equivalent expressions. It is thus important to abstract away such intermediary code that, if included in the behavior specification, would make the specification restricted to a particular malware variant.

The mining algorithm starts, at **Step 1**, by executing the malware (in a contained environment, of course) and one or more benign programs. We collect system-call traces during each execution. Each trace provides the system calls occurring on a particular path through the program code. For example, any execution of Bagle.J that reaches the code in Figure 1 produces a trace that includes a sequence of system calls similar to that shown in Figure 2(a). Unfortunately, we do lose information about the dependences between system call arguments.

Step 2 of the algorithm recovers some of the dependences present in the trace by using the actual argument values together with the corresponding type information. For example, in the trace of Figure 2(a), the third argument of RegCreateKeyA is a value returned by the system call (i.e., an out-argument). This argument has the same type (HANDLE) and the same value as the first argument of the subsequent call to RegDeleteValueA, which is a regular function argument (also called an in-argument). Thus, a def-use dependence [21] can be inferred between RegCreateKeyA on line 2 and RegDeleteValueA on line 3 (see Figure 2(b)). The process of inferring dependences between system call arguments is explained in more detail in Section 4.

The result of inferring dependences between system calls in Step 2 is a dependence graph, part of which is illustrated in Figure 2(c). This graph characterizes the relationships between the system calls observed during malware execution and was constructed without any disassembly, decompilation, or static analysis of the binary code. Each node represents one system call together with its arguments. Each edge represents dependences between arguments of different system calls. For example, the edge labeled $Y_1 = X_3$ denotes the fact that the first argument of RegDeleteValueA has the same value as the third argument of RegCreateKeyA.

Note that the actual argument values that appear in the trace have been removed from the dependence graph, both for constrained arguments (i.e., part of a dependence) and for unconstrained arguments. The values of constrained arguments are removed because, fundamentally, the fact that an argument is constrained is more important than its actual value, which can vary between executions and between variants. The values of unconstrained arguments are not significant because the arguments themselves are not significant.

We know that the obtained dependence graph is unique as it describes the operations performed by a given malware sample. Any program with the same dependence graph must

be as malicious as our malware sample because its interaction with the operating system cannot be distinguished from the malware sample. While we could use the dependence graph as a specification of malware behavior, this graph can be extremely large, with millions of system calls and with many operations that are not malicious *per se*. We would like to obtain a smaller specification of malicious behavior. To this end, in **Step 3** we trim the malware dependence graph by contrasting it with dependence graphs of benign programs. The result is a subgraph of the malware dependence graph that is not a subgraph of any one benign dependence graph. The graph obtained in this fashion is a specification of malicious behavior that one can use for malware detection.

3. SPECIFYING MALICIOUS BEHAVIOR

Based on our example from Section 2, we have the following three requirements of our specification language for describing malicious behavior.

Requirement 1: A specification must not constrain truly independent operations in any way. We observe that some high-level operations that constitute the malware behavior can be performed in any order. For example, in Figure 1 the call to RegDeleteKeyA and its associated argument-setup instructions on lines 4 and 5 can be performed at any point in the code sequence. This is due to the fact that this operation is independent from the other calls and thus any permutation of these operations will lead to the same end goal. Thus, the order in which any operations appear in a particular malware instance is not indicative of the true dependences between these operations and should not be reflected in the specification.

Requirement 2: A specification must relate dependent operations. The second requirement for a model of malicious behavior is to reflect correctly the true dependences that constrain the ordering of program operations. Simply eliminating all dependences between system calls in a specification would make the specification so generic that it would match both malicious and benign programs. Some dependences are due to data-flow and API-usage rules. For example, in Figure 1 two dependences relate the registry key handle defined by RegCreateKeyA with the registry key handles used by RegDeleteValueA and RegCloseKey, respectively. Other dependences are derived from the protocols that control interactions with other processes and systems external to the malware. For example, Bagle.J worm uses the SMTP protocol to send email. As a result, the arguments to network system calls such as send have to follow the rules of SMTP.

Requirement 3: A specification must capture only security-relevant operations. A specification of malicious behavior should not depend on operations that do not affect the trusted computing base (TCB), which is the operating system in our case. As a result, we want the specification to include only the security-relevant operations (i.e., system calls). The malicious behavior we capture describes the set of operations the malware performs on the TCB. For example, the Bagle.J backdoor communicates over the network using multiple system calls. The model for this backdoor functionality would include only the network-related system calls with dependences relating the values of their arguments, but with no additional constraints on how these values are computed internally by the malware.

Type	*Description*
HANDLE	Handle to an OS object.
`int`	Integer value.
string	C-style string value.
$\langle L_1 : \tau_1, \ldots, L_n : \tau_n \rangle$	Labeled n-tuple.
$\langle \tau_1 \vert \ldots \vert \tau_n \rangle$	Union.
dir τ	Direction, i.e., `in`, `out`, or `inout`.

Table 1: Type system for system-call arguments.

Here, we equate security-relevant operations with system calls. While not all system calls are security-sensitive, any system call can be part of a security-relevant behavior. For example, querying the name of the executable file for the current process is innocuous by itself, but can be used to implement self-replicating behavior. Thus, we do not restrict ourselves from the outset to particular system calls. Only after applying the mining algorithm we eliminate system calls that are not relevant in a security context.

We define a specification of malicious behavior as a dependence graph of system calls. Let Σ be the set of system calls. A system call $S \in \Sigma$ is a function of N variables, $S : \tau_1 \times \cdots \times \tau_N \to \tau_R$, where τ_i is the type of the i-th argument of the operation, and τ_R is the return type. The type system consists of an opaque handle type (for referencing various kinds of OS objects), an integer type, and a string type, together with a type constructor for creating labeled-tuple types, a type constructor for argument-direction type qualifiers, and a type constructor for unions (see Table 1).

Dependences between system calls are encoded as logical formulas constraining the values of the system-call arguments, e.g., $Y_1 = X_3$ in Figure 2(d). Additionally, system calls in the dependence graph can be constrained using local argument constraints. For example, the node RegDeleteKeyA in Figure 2(d) has two local constraints. Any logic with support for modular and bit-vector arithmetic, arrays, and existential and universal quantifiers is sufficient to represent our constraints on system calls. Let $\mathcal{L}_{Dep}$ be such a logic and let *Vars* denote the set of variables that appear in formulas of this logic. The set $\Sigma \times 2^{Vars}$ represents the set of symbolic system calls, i.e., system calls that have as arguments the uninterpreted variables from *Vars*.

DEFINITION 1 (MALSPEC). *A malicious specification is a directed acyclic graph (DAG) with nodes labeled using system calls from an alphabet Σ and edges labeled using logic formulas in a logic $\mathcal{L}_{Dep}$. Formally, the malicious specification (malspec) $\mathcal{M}$ is written $\mathcal{M} = (V, E, \gamma, \rho)$, where:*

- *V is vertex-set and E is edge-set, $E \subseteq V \times V$,*
- *γ associates vertices with symbolic system calls, $\gamma : V \to \Sigma \times 2^{Vars}$,*
- *ρ associates constraints with nodes and edges, $\rho : V \cup E \to \mathcal{L}_{Dep}$.*

For a node in $v \in V$ with label $S = \gamma(v)$, let us denote by S both the node and its system-call label when the context allows it unambiguously. We write $\langle n_1, n_2 \rangle \in E$ for an edge from node n_1 to node n_2 in the malspec and $n_1 \to^* n_2$ for a path from n_1 to n_2.

We note that our malspec language satisfies the requirements described earlier. Independent system calls (Requirement 1) can be represented in a malspec as disconnected

nodes, with no particular execution order. Dependent operations (Requirement 2), on the other hand, are represented as nodes connected via constraints in the graph. The third requirement is satisfied simply by the fact that vertices in the graph are system calls. No other syntactic elements of a binary program (e.g., instructions or offsets) are part of a malspec, thus rendering any malspec-based detector resilient to obfuscation attacks.

Expressive Power of Malspecs. Our design choice for the malspec language leads to several restrictions. Fundamentally, this malspec definition allows us to describe any malicious behavior that can be expressed as a safety property (i.e., a program is malicious if it performs a particular sequence of operations). As a result, this type of malspec cannot encode malicious behavior that results in the violation of some liveness property of the system (e.g., a denial of service attack). Currently, such a limitation might be of purely theoretical interest, because in our experience all current malware classes can be described in terms of safety violations.

A second limitation arises from our envisioned use of such malspecs. A program would match a malspec if at least one of its paths invokes the system calls of the malspec with arguments that satisfy the constraints of the malspec. Thus, a malspec is not designed to represent malicious behavior that encompasses multiple distinct executions of the program and in particular the use of covert channels, which consist of observable differences between multiple program executions. While a detector could interpret malspecs as describing constraints over multiple executions, this specification language might not be the best tool for such a purpose. We note that a malspec can still represent explicit information flows, which should cover most spyware in existence.

Finally, there is the question of whether all malicious behaviors can be described in terms of system calls. We note that system calls are the basic units of interaction between a program and its execution environment, the operating system. There are certainly other ways for a malicious program to impact a host, e.g., by influencing another program through shared memory, and we plan to investigate these behaviors in the future.

4. MALSPEC-MINING ALGORITHM

We now describe in detail the malspec-mining algorithm introduced in Section 2, with emphasis on the dependence-graph construction and the graph differencing steps.

The algorithm is based on dynamic analysis techniques that use execution traces obtained by running malware and benign programs in identical environments. The inputs to the algorithm are a malicious program and a set of benign programs. No additional information about the malicious program is necessary.

Pseudo-code for MiniMal is given in Algorithm 1. Lines 3 and 4 correspond to the first step introduced in Section 2, the collection of execution traces for both the malware variant and a set of benign programs. The second step of constructing the dependence graphs for each collected trace is implemented in the loop at line 6, while the malspec computation takes place inside the loop at line 11. We describe each of these steps in more detail in the rest of the section.

Step 1: Collect Execution Traces.

A representative trace must exhibit some malicious behavior for our algorithm to function. If there are no distinctions between the executions of the malicious program and of the benign programs, then no malicious behavior can be identified. In practice, getting the malware to be malicious is not difficult, as most malware tries to infect and spread in as many environments as possible.

The traces are collected by passively monitoring the execution of each program in a contained environment that is similar to a computer fully connected to the Internet. We describe the trace-collection environment and procedures in detail in Section 5.

Step 2: Construct Dependence Graphs.

There are three types of dependences that relate system calls to each other in a dependence graph. A *def-use dependence* expresses that a value output by one system call is used as input to another system call and is similar to the concept of def-use dependence from program analysis [21]. An *ordering dependence* between two system calls states that the first system call must precede the second system call. Ordering dependences can be rooted in API specifications (e.g., no calls to write after a call to close) or protocol specifications, where the commands sent and received over a communication channel must follow a prescribed sequence. Because inferring ordering dependences requires the availability of external specifications, our mining algorithm does not produce such dependences at this time. A *value dependence* is a logic formula expressing the conditions placed on the argument values of one or more system calls. Value dependences describe any non-trivial data manipulations the program performs in between system calls.

From the perspective of a malware writer that tries to create code variations with the same observable behavior, the only constraints on the code he writes arise from dependences directly related to the system calls implementing the observable behavior. This observation supports our intuition that argument values are not relevant unless part of a dependence. Any gap between the dependences envisioned by the malware writer and the dependences for which a malware detector searches represents an opportunity for obfuscation (if the dependences used in detection are stronger than those of the malicious behavior) or for false positives (if weaker).

The algorithm we present here computes an underapproximation of the dependence graph. The algorithm produces only a subset of def-use dependences, ignores all ordering dependences, and produces a limited set of value dependences. By underapproximating the malware dependence graph, we run the risk of not finding any differences between a particular malware instance and a benign program. This problem can be fixed by adding relevant dependences to the malware dependence graph, under the guidance of a human expert. On the other hand, the benefit of underapproximating the dependence graph is that any difference we find between the malware and any benign program is characteristic of all possible malware variants that share observable behaviors.

Def-Use Dependences. To discover def-use dependences between events (system calls) in an execution trace, we use argument values together with type information. Each argument of a system call has its type enhanced with a three-valued type qualifier that specifies whether the argument is an in argument, an out argument, or an inout argument.

MiniMal creates a dependence edge between two system

Input: A malware M and a set of benign programs $\{B_1, \ldots, B_K\}$.
Output: A set of malspecs $\{\ldots, \mathcal{M}_j, \ldots\}$.

begin
  /* Collect execution traces. */
  $t_M \leftarrow \mathit{ExecutionTrace}(M)$;
  for i=1 **to** K **do** $t_i \leftarrow \mathit{ExecutionTrace}(B_i)$;
  /* Create dependence graphs, one per trace. */
  foreach $t_x \in \{t_M, t_1, \ldots, t_K\}$ **do**
    $V \leftarrow \mathit{Events}(t_x)$;
    $E \leftarrow \mathit{InferDependences}(t_x)$;
    $G_x \leftarrow (V, E)$;
  /* Compute malspecs from components of G_M. */
  foreach $H_j \in \mathit{UniqueComponents}(G_M)$ **do**
    if $\mathit{IsTrivialComponent}(H_j)$ **then continue**;
    $\mathcal{M}_j \leftarrow (\emptyset, \emptyset)$;
    /* $\uplus$ = maximal union, $\ominus$ = minimal contrast subgraph. */
    for $i \leftarrow 1$ **to** K **do** $\mathcal{M}_j \leftarrow \mathcal{M}_j \uplus (H_j \ominus G_i)$
  return $\{\ldots, \mathcal{M}_j, \ldots\}$;
end

Algorithm 1: MiniMal

calls when the later system call (in execution-trace order) has an in (or inout) argument with the same type and the same value as the out (or inout) argument of the earlier system call.

The direct application of this rule to an execution trace leads to the creation of a large number of false def-use depences, because the majority of system calls have arguments with integer values. A integer value will often appear as an argument to unrelated system calls, although no such dependence was intended. To prevent this problem, we restrict the type of variables on which we check for def-use dependences to resource identifiers (HANDLEs in Microsoft Windows). Because the lifetime and state of resources are managed by the OS through well-defined APIs, we can correctly determine when two arguments identify the same resource. Even when the OS reuses a resource handle, we can disambiguate the use of the first handle from those of the second handle because they are separated by a call to deallocate or release the first handle (e.g., a call to NtClose).

Aggregation. As part of the def-use dependence computation, we also combine consecutive system calls that operate similarly on the same resource. Intuitively, one read of 1000 consecutive bytes from a file is equivalent to one thousand consecutive reads of 1 byte each. In particular, consecutive file operations of the same type (all reads or all writes) and network operations (all sends or all receives) are collapsed into one aggregate graph node.

The gain from aggregation is twofold. First, the vertex count of the dependence graph (initially the same as the size of the execution trace) can drop dramatically, reducing memory requirements and speeding up the later steps. Second, this aggregation of identical operations affords us some simple form of equivalence between system-call sequences (e.g., when a malware sample sends one byte at a time over the network, while most benign programs send information in larger chunks). The virus MyDoom.E is an example of such behavior, as it sends and receives network data one byte at a time.

Formally, we combine malspec nodes n_1 and n_2 if:

- n_1 and n_2 share an def-use predecessor, i.e., $\exists n. \rho(\langle n, n_1 \rangle) = (X = X_1) \wedge \rho(\langle n, n_2 \rangle) = (X = X_2)$,
- n_1 and n_2 are labeled with the same system call, i.e., $\gamma(n_1) = \gamma(n_2)$, and
- there is no node in the trace between n_1 and n_2 that performs a *different* system call on the same resource, i.e., $\nexists n' \in V. n_1 \rightarrow^* n' \rightarrow^* n_2 \wedge \rho(\langle n, n' \rangle) = (X = X') \wedge \gamma(n') \neq \gamma(n_1)$.

Substring Dependences. We address the issue of recovering dependences between system call events when their arguments are related through some computation, without being equal in value. We choose to focus on strings because, first, the vast majority of system calls have at least one string argument. There is a large amount of data that is passed between a program and the operating system in the form of strings or byte arrays. Second, in many cases the manipulation of a string preserves part of its original value.

We apply the following heuristic to discover string dependences. For each string-valued out (or inout) argument, we compare its value with the string-valued in (or inout) arguments of all its trace successors. If the two values share a substring of length greater than a threshold, then a dependence edge is added from the first system call to the second. The threshold can be customized. In our experiments, the threshold was set to 12, which was useful in minimizing any false dependences.

Self-Referential and Other Local Dependences. We notice that malware often reads from its own executable file during self propagation, while benign programs only read from other files. To capture this difference in behavior, we need to be able to classify file operations based on the file they affect (access to own executable file vs. access to any other file). Given the name of the executable image of the current process, we add self-referential dependences to all system calls that refer to that file.

We also recover a limited amount of dependences when the values of system call arguments are well-defined constants (e.g., HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE denote roots of particular Microsoft Windows registry hives).

Step 3: Compute Contrast Subgraph.

The previous two steps transform the execution traces of malware and benign programs into corresponding dependence graphs. The malspec we wish to identify is a subgraph of the malware dependence graph that does not appear in any of the benign dependence graphs. The simplest solution would be to choose the whole malware dependence graph, but the resulting malspec would be too large and too specific to the malware sample. Thus, we need to generalize the malspec by making it as small as possible.

In data mining a *minimal contrast subgraph* of two graphs is a smallest subgraph of the first graph that does not appear in the second. Formally, a contrast subgraph of two graphs G_1 and G_2 is a subgraph of G_1 that is not subgraph-isomorphic to G_2. A contrast subgraph is minimal if none of its subgraphs is a contrast subgraph. Then we can compute a minimal malspec as a minimal contrast subgraph of a malware dependence graph and a benign dependence graph.

An algorithm for minimal contrast subgraphs has been proposed Ting and Bailey [26] and is denoted by the $\ominus$ operator in the pseudo-code of Algorithm 1.

Before we apply the Ting-Bailey algorithm, we make the observation that we are only interested in malspecs that are connected. This is because any disconnected malspec, i.e., any disconnected minimal contrast subgraph, is composed of a set of actions repeated many times (for a formal proof, please see the appendix at http://www.cs.wisc.edu/~mihai/publications/minimal-appendix.pdf). For example, if a malware opens and reads 10 files, while a benign program opens and read 5 files, then a malspec could consist of opening and reading 6 files. Because the repetition count is under the control of the attacker, we are not interested in disconnected malspecs.

A malspec then has to be connected, allowing us to optimize the search for minimal contrast subgraphs by limiting it to individual components of the malware dependence graph. We apply the minimal contrast-subgraph mining algorithm to each component of the malware dependence graph and a benign dependence graph. The result is a collection of malspecs, one per component of malware depedence graph, where each malspec characterizes a unique malicious behavior.

Furthermore, not all components of the malware dependence graph are of interest. For example, a set of queries about system parameters do not signal malicious intent, even though such queries might only appear in malicious programs. We eliminate such components from further analysis using the function *IsTrivialComponent*, which identifies graphs that are not interesting from a security perspective. Only interesting graph components are processed by the minimal contrast subgraph algorithm.

The contrast-mining algorithm needs to determine when nodes are equivalent and when edges are equivalent. We use the following definitions of node equivalence and edge equivalence. For two nodes, n_1 from the malware dependence graph and n_2 from a benign dependence graph, we say that they are equivalent, written $n_1 \equiv n_2$, when they have the same system-call label and the constraint associated with the benign node implies the constraint associated with the malicious node, $n_1 \equiv n_2 \iff (\gamma(n_1) = \gamma(n_2)) \wedge (\rho(n_2) \Rightarrow \rho(n_1))$. Similarly, for two edges, e_1 from the malware dependence graph and e_2 from the benign dependence graph, we say that they are equivalent, written $e_1 \equiv e_2$ if the constraint associated with the benign edge implies the constraint associated with the malicious edge, $e_1 \equiv e_2 \iff (\rho(e_2) \Rightarrow \rho(e_1))$. Note that these notions of equivalence, although not reflexive or symmetric, suit us nonetheless because of the asymmetry of the contrast subgraph operation.

To ensure that a malspec does not lead to false positives, we compare a malware dependence-graph component with a set of more than one benign dependence graphs. Because two different benign programs might produce different minimal contrast subgraphs, the resulting malspec must be the union of all minimal contrast subgraphs. If we use $\biguplus$ to denote maximal union, i.e., the union without any graphs which are subgraphs of others in the set, then we can compute the malspec as shown in line 15 of Algorithm 1.

5. EMPIRICAL EVALUATION

In a series of experiments we performed to validate our algorithm, we mined malspecs for 16 well-known malware samples. In each case, the algorithm successfully discovered the same behavioral features as those independently provided by human experts. We compared MiniMal's results with specifications created by Symantec's virus analysts to determine how descriptive the mined malspecs were in comparison. Additionally, we explored the use the mined malspecs for the semantics-aware malware detector of Christodorescu *et al.* [7], to establish the applicability of our mining technique to existing detection techniques.

Experimental Setup. The evaluation environment consisted of one computer acting as the victim machine and a second computer acting as any number of machines on the Internet. The victim machine, where the malware was run, used Microsoft Windows 2000 Professional as the operating system. Several common applications were additionally installed (Mozilla Firefox, Mozilla Thunderbird, Adobe Acrobat Reader, Microsoft Outlook Express), together with the software for collecting system-call traces (an updated version of BindView's strace for Windows [5]). This configuration formed the baseline for our experiments–every program whose system-call trace we collected was started in this environment.

The malicious programs we considered in our experiments propagate over the Internet and require an Internet connection to activate their payloads. We used the second machine in our evaluation environment as "the rest of the Internet" from the point of view of the victim machine. This machine was configured to respond to packets sent from the victim machine to any IP address and also to simulate several common Internet services (web server, SMTP, POP, and CIFS/Samba), similar to the Internet-inna-box work [28].

Malware Sample Set. We evaluated 16 malware samples, as follows: Netsky variants A through J, Bagle variants A, C, and J, and MyDoom A, E, and G. These malware samples were chosen because they exhibited the malicious behavior described on anti-virus web sites inside our evaluation environment. This way we ensured that each of these samples produced malicious behavior relevant for mining. For each sample we obtained execution traces of 1, 2, and 4 minutes in duration. Statistics for some of the program traces and the dependence graphs derived from them are listed in Table 2.

Benign Program Set. The following programs were part of the benign program collection: Mozilla Firefox 1.5.0.6, Mozilla Thunderbird 1.5.0.5, Microsoft Outlook Express 5, and the installation programs for Mozilla Firefox 1.5.0.6, Mozilla Thunderbird 1.5.0.5, and MikTeX 2.5. These programs were chosen for their behavioral similarity with the malware samples used. Firefox, Thunderbird, and Outlook Express generated network communications, Thunderbird and Outlook Express sent email messages, and the installation programs accessed Windows registry keys and installed files in system directories.

Each benign program was executed for 1 and 2 minutes (or until completion), enough to perform tasks common to each program. We manually provided input to the programs where appropriate (e.g., by pointing Firefox to the CNN website and then clicking on story links). Statistics for benign traces were listed in Table 2.

5.1 Quality of Mined Malspecs

We conducted an empirical evaluation of the malspecs obtained from MiniMal by comparing them with the be-

Sample	*Trace*		*Dep. Graph*	
	Time	*Size*	*Node #*	*Edge #*
Netsky.A	60 s	256710	162174	12578
Netsky.F	60 s	507536	427681	1182
Netsky.F	120 s	982986	780870	2184
Netsky.F	240 s	1903315	1513600	6136
Firefox	60 s	47502	45875	1692
Thunderbird	60 s	112415	78040	3001
Outlook Ex.	60 s	151023	141992	2184
MikTeX inst.	60 s	17253	17253	348

Table 2: Test set statistics.

Expert Description of Features	*Malspecs*	
	Description	*Syscalls*
Create a mutex.	*same*	2
Copy self to system dir.	Write file to system dir.	192
Set registry key to run self at boot.	Set registry key to run file from sysdir at boot.	3
Delete registry keys for AV tools.	*same*	7
Copy self to system directory as ZIP file.	-	-
Search for email addresses and email self.	Communicate via email.	143272
Copy self to net drives.	-	-
-	Query net-related registry keys.	11272

Table 3: Comparison of expert-provided behavioral descriptions and mined malspecs from Netsky.A.

havioral features described by human experts. For domain expertise we relied on the malware descriptions that appear on anti-virus websites such as the Symantec Antivirus Research Center [24]. Such descriptions serve our purpose well as they reflect the results of manual analysis by a human expert and are given in terms of high-level features.

Tables 3 and 4 summarize our results. In Table 3, we examine in detail the results of malspec-mining one malware sample, Netsky.A. Symantec's description of Netsky.A contains seven high-level behaviors specific to the virus. Table 3 lists, in the left column, these high-level behaviors. On the right hand side we show the malspecs mined by MiniMal. For each malspecs we also show the number of system calls implementing it. MiniMal isolated five of the seven behavioral features identified by the human expert in Netsky.A. Additionally, the results contained 95 other malspecs specific to Netsky.A (including the one listed at the bottom of Table 3), encompassing various queries to system parameters and the Windows registry. These queries were identified as unique to Netsky.A because no benign program from our sample set performed such an exhaustive set of queries. The results show that our malspec-mining algorithm performs quite well in comparison to a human expert.

Two malspecs stood out as missing from the results mined out of Netsky.A. First, the virus copied itself to the system directory in ZIP-compressed form. MiniMal did not identify this behavior as suspicious because it could not connect the read from self with the write to the system directory. The missing link was the ZIP compression that transforms the input data (from the virus itself) into the output file (in the system directory). Since we did not recover the data dependences through this particular transformation, this behavioral feature was not captured in a malspec. The other missing feature was the self propagation through network

Malware Sample	*Expert-Provided Malspecs*	*Mined Malspecs*	
		Matching	*Total*
Netsky.A	7	5	100
Netsky.B	10	8	113
Netsky.C	10	8	65
Netsky.D	7	7	66
Netsky.E	6	5	48
Netsky.F	6	5	73
Netsky.G	6	4	91
Netsky.H	6	5	106
Netsky.I	5	4	66
Netsky.J	8	7	105
Bagle.A	8	6	89
Bagle.C	12	7	80
Bagle.J	8	7	102
MyDoom.A	7	5	115
MyDoom.E	10	7	110
MyDoom.G	12	7	88

Table 4: Summary of the malspec-mining results. The *Matching* column lists the number of mined malspecs that matched those from human experts.

drives. In this execution trace, the virus did not propagate through network drives, and thus MiniMal could not identify a malspec for this behavior.

Our automated algorithm obtained similar results for the other samples. A summary of the malspec-mining data is shown in Table 4, which lists for each malware sample the number of expert-provided malspecs, the number of matching mined malspecs, and the total number of malspecs.

We noted a large number of additional malspecs mined from each malware sample. Analyzing these, we determined that they were of no particular interest from a security perspective, as these malspecs fit into one of two categories. First, many of these malspecs captured information queries (e.g., reads from registry or from particular files). Second, the remaining malspecs encoded a basic operation repeated multiple times (e.g., multiple reads from distinct files), where the basic operation (i.e., reads from a file) also appeared in benign programs but not as many times as in the malware. Both these classes of malspec could be safely eliminated from further analysis and from use in detection.

5.2 Malware Detection Using Malspecs

The preceding results supported our hypothesis that distinctions between malicious and benign programs can be captured as differences in system-call depedences, inferred from execution traces. We were also interested in determining how easy it was to use a mined malspec in an existing detector and how many malware variants matched a malspec.

We chose the *semantics-aware malware detector* [7] as testbed for our experiments. This detector employs a combination of static analyses and decision procedures to decide whether a program matches a specification of malicious behavior called a *template*. This template of malicious activity was a graph of machine instructions with uninterpreted variables as arguments, together with equality constraints on these variables.

We used a enhanced version of the semantics-aware malware detection prototype that can analyze a program together with its supporting dynamically linked libraries and can check arbitrary constraints on uninterpreted variables. A malspec is then mapped to a semantics-aware template:

1. We converted system-call identifiers to corresponding instruction sequences. For example, on Microsoft Windows XP with Service Pack 2, the system call NtCreateFile was translated to the sequence (X is an uninterpreted variable):

```
1 mov   eax, 25h
2 mov   X, 7FFE0300h
3 call  dword ptr [X]
```

2. We converted malspec constraints to corresponding template constraints. In particular, due to calling conventions, system-call arguments from the malspec were replaced with stack locations.

The end result was one semantics-aware template for each malspec we mined.

Our testing strategy had two steps. We used MiniMal to derive malspecs from early variants of several malware families (Netsky, Beagle, MyDoom). Then, we tested subsequent malware variants against the corresponding malspecs using the semantics-aware detector. Because of limitations in the current version of the detector, related to the static analysis of indirect call targets, we were unable to test all variants in our test set successfully. Nonetheless, four variants of Netsky matched the self-installation malspec mined from Netsky.A. This indicates that mined malspecs could provide forward detection. We plan to further evaluate this approach as soon as the detection prototype stabilizes.

5.3 Discussion

The evaluation showed that our malspec-mining algorithm was successful in identifying behaviors unique to malware samples. It did so without any *a priori* security policy that defined malicious or benign behaviors. Furthermore, these malspecs could then be used to provide forward detection of later malware variants.

There were several aspects to address when considering the validity of our experiments. The most important one was the internal validity, i.e., the factors that could influence the observed behavior of the test programs (both benign and malicious). We considered two sources of threats to internal validity: (1) impact of execution monitoring, and (2) relevance of execution traces. The execution monitor simply collected system-call traces for a fixed time period. Since the monitored program could not observe the execution monitor, which was present only in protected kernel space, we expected the monitored program to execute normally, without biasing our results.

The threat of execution relevance arose from differences between our "emulated Internet" environment and the real Internet. If the malware attempted, for example, to download additional components or commands from a remote server, it would fail without executing any malicious behaviors. Similarly, the malware might not execute any malicious behaviors during the monitoring period if it was time-dependent. In such cases, MiniMal would not discover any differences between malicious and benign programs. We operated under the assumption that most malware tries to execute in as many environments as possible. In future, we will explore ways to eliminate this assumption, for example by allowing limited interaction with the Internet and by analyzing multiple execution paths [20].

A second aspect was that of external validity, the lack of which could limit our ability to generalize our malspec-mining technique to other classes of malicious programs. While our experiments focused on mass-mailing worms, the technique was general enough to apply to any class of malware behaviors. In particular, we note that the variants in our test set exhibited behaviors above and beyond those traditionally characteristic of a mass-mailing worm.

Finally, the third aspect of concern was that of construct validity. This relates to how expressive the malspec language is and how powerful the dependence graph construction is. We discussed the malspec language in Section 3. As noted in the results above, the dependence inference failed when the information flow between system-call arguments was complex. We plan to address this limitation in future work, possibly using dynamic information-flow tracing.

6. RELATED WORK

There is a large body of work in the area of learning program features and program understanding using dynamic traces. We focus on two areas most related to our work, specification mining for software engineering and specification mining for software security.

Mining of specifications has received considerable attention for a variety of program-related aspects, including API-usage and general programming rules [1, 18, 27]. In contrast, we mine specifications of malicious behavior, with no *a priori* definition of "malicious," by comparing malicious and benign programs. Clone-detection techniques compare programs [3,11,13,30], but they target similarities, not distinctions, between and within programs. Program-differencing techniques have goals similar to ours, but are limited to distinctions at the statement level or at the structural level [2, 9, 10, 17]. The execution-history matching introduced by Zhang and Gupta [30] is closest to our technique in that it analyzes dynamic data-dependence graphs, albeit with the goal of finding similar programs. Execution-history matching also operates at the instruction level and requires precise information about data and control dependences, and thus it is not directly applicable to our problem.

In computer security, the area of host-based intrusion detection has applied specification mining techniques to derive models of program behavior, which can be used to contain execution. These program models of "good" behavior can be derived through static analysis [8] or learned from dynamic traces [23]. Recent work has looked at the use of constraints over system-call arguments for host-based intrusion detection. Kruegel *et al.* proposed a system for learning models for the string values of individual system-call arguments without taking into account dependences between system calls [15]. Their models could be used as local constraints in the malspecs we extract. Bhatkar *et al.* introduced a technique for deriving a dataflow model to be used for intrusion detection [4]. They derived both local constraints and dependences for system-call arguments, using a fine-grained set of relations over strings (e.g., isWithinDir, hasExtension, hasSameDirAs). These types of constraints fit into our malspec language and we will research extending our dependence-graph construction to discover such string constraints.

7. CONCLUSION

We introduced a language for specifying malicious behavior and an algorithm for mining such specifications (mal-

specs) from dynamic traces of malware samples and benign programs. Our prototype, MiniMal, infers malspecs by differencing the dependence graphs of a malware sample and of multiple benign programs. Experimental results showed that the malspecs mined by our algorithm compare favorably with behavioral specifications manually constructed by human experts. Malspecs can also be used to detect multiple malware variants.

The current work computed the behavioral difference as a subgraph of the malicious dependence graph, without taking into account the constraints of any similar benign subgraph. One possibility is to define the difference to consider both the malicious constraints and the (negation of) benign constraints — we plan to explore this in future research.

The choice of test programs is an important element of our mining approach. For malicious program we would like to know whether certain classes of malware are more amenable to mining. For benign programs we want to ensure that a broad selection of behaviors are represented in the test set. Thus a goal for future work is to determine the impact the choice of test programs has on the quality of the mined malspecs. A related problem is the incompleteness of the dependence graph built from a finite set of execution traces. Combining dynamic and static analysis could provide techniques to derive a better dependence graph.

8. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 4–16, 2002.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, Sept. 2004.

[3] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 156–165, New York, NY, USA, 2005. ACM Press.

[4] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proc. IEEE Symposium on Security and Privacy*, pages 48–62, 2006.

[5] BindView. Strace for NT. Published online at http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm (accessed 9 Sep. 2006).

[6] M. Christodorescu and S. Jha. Testing malware detectors. In *ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04)*, pages 34–44, 2004.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[8] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proc. 11th Network and Distributed System Security Symposium (NDSS'04)*, 2004.

[9] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI'90)*, pages 234–245, 1990.

[10] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. International Conference on Software Maintenance (ICSM'94)*, pages 243–252, 1994.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[12] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proc. 2nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05)*, pages 174–187, 2005.

[13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. 8th International Symposium on Static Analysis (SAS'01)*, pages 40–56, London, UK, 2001. Springer-Verlag.

[14] C. Kruegel, D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Reverse engineering of network signatures. In *AusCERT Asia Pacific IT Security Conference*, 2005.

[15] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS'03)*, pages 101–118, 2003.

[16] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, 2004.

[17] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proc. Conference on Software Maintenance*, pages 282–290, Nov. 9–12 1992.

[18] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 306–315, New York, NY, USA, 2005. ACM Press.

[19] A. Marinescu. Russian doll. *Virus Bulletin*, 15(8):7–9, Aug. 2003.

[20] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[21] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, Jan. 1997.

[23] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[24] Symantec Antivirus Research Center. Expanded threat list and virus encyclopedia. Published online at http://www.symantec.com/enterprise/security_response/threatexplorer/index.jsp (accessed 9 Sep. 2006).

[25] P. Ször and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, pages 123 – 144, 2001.

[26] R. M. H. Ting and J. Bailey. Mining minimal contrast subgraph patterns. In *6th SIAM International Conference on Data Mining*, pages 638–642, 2006.

[27] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 461–476, 2005.

[28] I. Whalley, B. Arnold, D. Chess, J. Morar, and A. Segal. An environment for controlled worm replication & analysis (Internet-inna-Box). In *Virus Bulletin Conference*, 2000.

[29] z0mbie. z0mbie's homepage. Published online at http://z0mbie.host.sk (accessed 16 Jan. 2004).

[30] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 197–206, 2005.