Towards Discovering and Containing Privacy Violations in Software

Louis Kruger  
Computer Sciences Department  
University of Wisconsin  
Madison, WI 53726  
lpkruger@cs.wisc.edu

Hao Wang  
Computer Sciences Department  
University of Wisconsin  
Madison, WI 53706  
hbwang@cs.wisc.edu

Somesh Jha  
Computer Sciences Department  
University of Wisconsin  
Madison, WI 53726  
jha@cs.wisc.edu

Patrick D. McDaniel  
Pennsylvania State University  
State College, PA 16801  
mcdaniel@cse.psu.edu

Wenkee Lee  
College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332  
wenke@cc.gatech.edu

September 17, 2004

Technical Report  
UW-CS-TR-1515

Abstract

Malicious code can wreak havoc on our cyberinfrastructure. Hence, discovering and containing malicious code is an important goal. This paper focuses on privacy-violating malicious code. Examples of privacy violations are leaking private user data to an external entity or downloading data to a user’s host without their permission. Spyware, which has recently received considerable attention in the popular literature, is an important example of privacy-violating malicious code. We propose a multi-step approach to discovering and containing privacy violations. We have designed and implemented a dynamic slicing tool to discover dependencies between events in an execution trace. We demonstrate that dynamic slicing can be used to discover privacy violations. Information gathered using dynamic slicing can be used to construct security policies to contain the discovered privacy violations. These security policies are then enforced by a sandbox. We have implemented a sandbox for Windows, and have successfully evaluated our approach on two applications: KaZaa and RealOne Player. For both of these applications we were able to discover privacy violations in them using our dynamic-slicing tool. Moreover, using information gathered through dynamic slicing we were able to design policies to thwart these privacy-violations. Although our preliminary evaluation was performed on spyware, in the future we will evaluate our approach on other privacy-violating malicious code.
1 Introduction

Malicious code can infiltrate hosts using a variety of methods, such as attacks against known software flaws, hidden functionality in regular programs, and social engineering; a taxonomy of malicious code appears in [35]. Disruption of services caused by malicious code can have catastrophic effects, including loss of human life, disruption of essential services, and huge financial losses. The increased reliance of critical services on our cyberinfrastructure and the dire consequences of security breaches have highlighted the importance of detecting and containing malicious code.

This paper addresses privacy-violating malicious code, where a privacy violation is defined as the disclosure of private information or use of a resource without its owner’s permission or knowledge. Transmitting sensitive user data to an external server and downloading data to a user’s host without their knowledge are two concrete examples of privacy-violations. Spyware is an important example of privacy-violating malicious code. Spyware [52, 56] is malicious code that performs useful functionality, but results in disclosure of sensitive information, such as a customer’s credit card information. RealNetworks’ Real Jukebox [31, 43] was one of the well publicized examples of spyware. In [35] spyware is defined as follows:

Spyware is a useful software package that also transmits private user data to an external entity.

A recent measurement study discovered that 10% of active hosts on a campus of a major public university were infected with spyware [47]. Given the widespread prevalence of spyware and privacy breaches that can result from them, detecting and containing privacy-violations is an important goal. The initial focus of our work is on spyware, but in the future we will evaluate other privacy-violating malicious code. Discovering privacy violations can be viewed as a special case of information-flow analysis [14, 36, 44, 54]. Since we are analyzing programs without access to source code, we will have to perform information-flow analysis on large executables. To our knowledge, information-flow analysis techniques have not be demonstrated to scale to large executables. Dynamic slicing finds dependencies between events in an execution trace. Our methodology uses a combination of dynamic slicing (which can be thought of as information-flow analysis for execution traces) and sandboxing.

There are two major classes of techniques for addressing malicious code: static and dynamic. Static analysis techniques [6, 7, 11, 12, 24, 33] attempt to detect malicious behavior in programs. Typically, virus scanners search for a pattern of instructions and thus use static analysis (albeit a very simple one). Dynamic techniques confine the behavior of a potentially malicious program using sandboxing. A sandbox [15, 19, 23, 29] monitors a program and confines its execution to a security policy. However, these techniques have not been applied to contain privacy violations.

This paper makes the following contributions:

Discovering privacy-violations using dynamic slicing: Malicious code, such as a virus or a worm, leave an observable footprint behind. In contrast, privacy-violating malicious code (e.g., spyware) perform useful functionality, such as downloading or replaying music, but also perform stealthy privacy-violating activities. Therefore, discovering such privacy-violating features is challenging. We have designed and implemented a dynamic-slicing algorithm that enables an analyst to discover privacy-violating features using event traces. Unlike traditional dynamic-slicing algorithms [3, 30] our algorithm works with incomplete information (in our case, a sequence of Win32 API calls made by the application). A description of our dynamic-slicing algorithm and its use in discovering privacy-violations appears in Section 4. Recent work on mimicry and hiding attacks [50, 51, 55] has demonstrated that it is important to incorporate dependency and argument information in security policies used for enforcement. Dynamic slicing can be used to recover dependencies between events and recover argument values, which can be used to construct precise security policies.
Containing privacy-violations using sandboxing: We present the design and implementation of a dynamic-monitoring infrastructure for Windows, called sandBoX86 . We have designed a language called the event description language or EDL, to abstract events. EDL provides a decoupling between raw events (Win32 API calls in our case) and abstract events used for monitoring, which makes porting our architecture to other platforms easier. Security policies are defined using a language called the event sequence language or ESL, which uses the abstract events defined by a EDL specification. These security policies also use the information gathered using dynamic slicing. EDL and ESL specifications for an application are then compiled into a monitor. Details about the architecture can be found in Section 5.

Case studies: In order to evaluate our sandboxing system we considered two applications: KaZaa [25] and RealOne Player [41]. A detailed description of these applications along with privacy-violating features we discovered in them can be found in Section 6. Details about these features were discovered using our dynamic slicing tool. For these two applications, we were able to construct security policies to thwart the discovered privacy-violating features, which lead us to believe that sandboxing can be very effective in containing privacy-violating malicious code, such as spyware. We also measured the memory and execution overheads introduced by our sandbox. Our evaluation demonstrated that the performance and resource costs associated with sandBoX86 are nominal. The macrobenchmarks show a slow down of less than 3.2% for all our applications. The results of these experiments can be found in Section 7.

2 Overview

This section gives an overview of our entire methodology. We describe how dynamic slicing can be used to discover privacy violations and gather information for constructing security policies. We also give two examples of security policies in ESL.

2.1 (Step 1) analyzing execution traces using dynamic slicing

Dependencies between events in an execution trace can be inferred using dynamic slicing. These dependencies can then be used to discover privacy violations and gather information to construct security policies. We will illustrate the use of dynamic slicing using two scenarios.

Scenario 1 (discovering privacy violations): A security analyst wants to discover whether a network event (e.g. sending a message to a remote host) depends on a sensitive event (e.g. reading from a specific file). This dependency indicates that the application is sending data obtained from a sensitive operation to a remote host.

Analysis: A forward slice from a sensitive event shows all the events that depend on it. If a network event is in the forward slice, it represents a potential privacy violation.

Scenario 2 (information gathering): Suppose a security analyst wants to discover all the remote hosts that an application connects to. This information will be useful in recognizing servers that are used by certain applications to download pop-up advertisements or to send information about user activity.

Analysis: Figure 1 demonstrates information gathering using dynamic slicing. The left side of the figure lists a portion of a program’s running trace. The right side shows the results of dynamic slicing. In this example, we want to find all the hosts which a program has sent data to using the send call. First, we isolate the send events. In this case there are two: 170 and 284. Next, backward slicing is performed from each one of the two send events. The result from the backward slicing consists of two trees, one from each of the two send events. From the two slices, we extract only the gethostbyname calls, which give us the set of hosts the
program has sent data to. Appendix B.2 shows a screen shot of our GUI interface to the dynamic slicer. The GUI interface provides a mechanism for a security analyst to visualize and browse the slices.

2.2 (Step 2) constructing security policies

In this step a security analyst designs ESL security policies to thwart privacy violations. Next we show two example policies which are related to scenarios 1 and 2 described before.

Example 1 Assume that using dynamic slicing an analyst was able to discover a privacy violation. Dependencies between events exposed during dynamic slicing helps an analyst construct a policy to thwart the discovered policy violation. Hence dynamic slicing can expose some of the context missing from traditional kernel-level enforcement mechanisms by detecting and tracking dependencies between operating system interface calls (e.g., API calls in windows, syscalls in UNIX). We demonstrate the power of this approach by describing the implementation of a canonical privacy policy: preventing the transmission of sensitive files across a network.

Consider a user who wishes to protect the files in her web browser cache from being exported by a malicious program. Pragmatically, she wants to prevent any process from reading items in her cache and then subsequently transmitting them over the network. This requires the system to infer dependencies between two very separate operations; the reading of data from the sensitive (cache) directory, and the use of network interfaces.

Figure 2 shows an ESL policy that protects the browser cache. Line 1 indicates that this ESL security policy uses the EDL specification given in file generic.edl. Lines 4, 5, and 6 define how the reading of sensitive material is detected. Line 4 defines a protection domain by identifying the sensitive cache directories for both Internet Explorer and Netscape, and lines 5 and 6 indicate how access to the protection domain should be monitored. The policy tracks the open and subsequent read of sensitive files by saving state between API calls. Specifically, the addHash operator allows open and read operations to be correlated with the file handle results of open calls (which are stored in the handles hash table).\footnote{Note that such mappings can be trivially circumvented by file descriptor aliasing operators such as dup. Note that these require OS API or system calls, and as such can be governed by policies in a similar manner as the open call. For brevity, we omit further discussion of these interfaces.}
the file descriptor mapping in the hash table, line 6 annotates (with a block flag) the process as having consumed sensitive data when it reads from a sensitive file. Note that prior to this annotation, the process can freely send and receive data over the network. Lines 7-11 state how the policy is enforced. If the block flag is true, then the argument corresponding to the host in the send call is changed to the localhost (lines 8-9). All other operations are permitted independent of annotation, e.g., connect, recv, recvfrom, etc.

While context mapping allows semantically deep policy enforcement, it is still limited to the information present in the interface calls themselves. In the normal case, this may lead to false-positives. For example, this example is sufficient to prevent arbitrary malicious software from performing these operations, but a normally behaving browser would frequently violate this policy and be prevented from using the network. During normal operation a browser frequently reads cache files and sends data. However, the cache data is not sent by the browser. The problem is that the policy does not have sufficient context to determine if the data being sent is derived from sensitive data. In general, determining data dependence requires information flow analysis, which is well known to be enormously difficult. For example, the cascade problem involves aggregation of authorized information flows to produce an unauthorized flow, which arises in networks of systems. The problem of removing such cascades is NP-complete \[21\]. While this work does not solve the most general problem (information flow), we argue that the semantically rich policies made possible using this approach represent a significant step forward in tracking and controlling the behavior of untrusted software.

**Example 2** Our second example is related to scenario 2. KaZaa downloads and displays advertisements during its use. Furthermore, KaZaa users cannot prevent KaZaa from displaying the advertisements unless they choose to uninstall it. Our goal, therefore, is to be able to disable KaZaa’s advertisement feature without having to uninstall KaZaa. We now give a policy that can detect and prevent KaZaa from connecting, sending or receiving any data from a list of blocked servers. At the same time, the policy allows other activities such as searching and downloading files from peer hosts. The policy is shown in Figure 3. Line 3 specifies the list of remote servers we want to block KaZaa from accessing. This list is constructed using dynamic slicing as was discussed earlier. In Line 4, we hash each IP address KaZaa has obtained through the gethostbyname call. Line 5 of the policy prevents KaZaa from establishing TCP connections to any of the remote servers that is on the blocked list. Lines 6 and 7 deal with the cases where KaZaa attempts to send or receive data from the blocked servers using UDP sockets.

This policy achieves the goal of stopping KaZaa from downloading advertisements from a list of remote
servers, which, consequently, prevents KaZaa from displaying the advertisements. The policy has one additional property: it also stops KaZaa from sending any information to these servers, which potentially could be more damaging since KaZaa can send private or sensitive data back to these servers. Line 6 stops this potential spying activity by intercepting any attempt to send data to the blocked servers. When KaZaa tries to send data to any one of the remote servers, we do not execute the send call but instead return success as if the call went through.

Above we give an example policy that stops KaZaa from receiving unwanted data from remote servers. We want to emphasize that this is just one example demonstrating the expressiveness of our policy language. In general, one can use ESL policies to block any sequence of events where the policy may seem to fit.

2.3 (Step 3) enforcing security policies

ESL security policies are enforced by our sandbox. We briefly describe the the process by which the security policies are enforced. Details can be found in Section 5. The ESL security policies described before are compiled into two dynamically loadable libraries (DLLs): event-interceptor and policy-enforcer DLL. A raw event (in our case of Win32 API call) is transformed into an abstract event by the event-interceptor DLL. The abstract event is passed onto the policy-enforcer DLL, which decides on the action to be taken for that event.

3 Related Work

This section considers how past works have detected and contained undesirable software behavior (e.g., bugs, spyware, malicious code).

Dynamic slicing algorithms typically reconstruct dependencies between statements in an execution trace [3, 4, 30, 53, 57]. Since slicing tracks the flow of values between statements, it can be cast in an information-flow framework [9, 14, 20, 44]. Agrawal et al. showed how dynamic slicing can be used to narrow the search for software errors, and hence reduce debugging costs [3]. Their dynamic slicing tool works backward from a known erroneous output toward the possible sources of the error. All code upon which the error state is dependent (as indicated by trace and symbolic information) is included in the program slice. Because all causality is captured in the slice, no other code need be searched for the source of the error. Smith and Korel recognized the difficulty and cost of obtaining and analyzing potentially large execution traces [49]. Also used for debugging, they define an algorithm that detects dependencies between logged resource usage events, rather than complete execution traces. Event slices are constructed from computed event dependencies and used to identify the cause of an error.
Slicing has also been shown to be a valuable tool for detecting malicious behavior. For example, King and Chen show how to reconstruct adversarial behavior from traces of operating system events on compromised hosts [27]. Their BackTracker system uses timed filesystem and process system call traces to reconstruct possible compromising events. BackTracker reconstructs a graph of relevant operations working from some known compromised behavior. This graph highlights the possible sources of compromise, and hence is a succinct and intuitive forensic representation of the attack. We note that such solutions may face similar issues as debugging slicer applications. For example, the cost of keeping complete execution or event traces may be prohibitive. We posit that sampling techniques, such as those proposed Liblit et al. [32], may help mitigate these costs.

Sandboxing is an increasingly important tool for secure systems deployment [15, 17, 18, 19, 23]. Sandbox tools prevent malicious behavior by monitoring and regulating security relevant operations according to some policy. For example, Provos regulates process operation by intercepting and enforcing policy over system calls [39]. These policies reduce the range of possible behaviors of a governed process, and hence the power of the compromising adversary. Garfinkel provides a thorough treatment of complex issues faced by designers of these system call interposition solutions [16]. Other systems are based on similar ideas, but built upon formal policy models [8, 45, 46]. Based on the Flask architecture, the SELinux platform uses a number of similar enforcement mechanisms to enforce role based access control, type safety, and multi-level security policy [34].

Policy enforcing solutions need not require privileged or kernel level access. Jain and Sekar introduce the notion of user level interposition of system calls as a vehicle for intrusion detection and malicious behavior mitigation [23]. Other solutions attempt to control program flow [28] or monitor for and prevent specific behaviors (e.g., buffer overflows) [13] through code instrumentation or process introspection.

Several commercial products have recently been launched in response to heightened public concern about spyware. In addition to a range of other security related features, Zone Alarm [58, 59] restricts local network connections based on a local policy. Zone Alarm further allows policy to be learned through user input, e.g., by appealing to the user when network operations with unspecified policy occur. Other commercial products treat spyware as viral behavior. Products such as Ad-aware [1, 2] develop signatures for spyware. Local drives, memory, and attachments are checked for the presence of spyware. Where found, the offending sources are quarantined.

While we build on many of these works, but the goals and execution of sandBoX86 make it unique. The novelty of our solution lies not only in the technology, but in the domain of policy; we focus on services that ensure user privacy. We use dynamic slicing to analyze behavior and guide policy construction, rather than as a forensic tool. Moreover, our dynamic slicing tool processes execution traces that do not have complete information (e.g., only system calls are visible but all the internal computation is hidden). The idea of interposing a library to intercept Win32 APIs has been suggested [22, 42] and UNIX system call interposition is well understood. However, we are the first to integrate comprehensive tools for policy creation and their subsequent enforcement. The following sections detail how we bring these elements together to form a complete and novel solution to combat privacy violations.

4 Dynamic Slicing

Dynamic slicing attempts to discover dependencies between events in an execution trace. In this section we present our dynamic-slicing algorithm. One of the main challenges we faced in dynamic slicing was incomplete information. For example, we only log events that correspond to Win32 API calls, and therefore all the information about the internal computation of the application is absent from the trace. We present a
4.1 Dynamic Slicing

This subsection provides the description of our dynamic-slicing algorithm. First, we formally define events and traces. Different events (such as gethostbyname and bind) use different types to represent the same information. For example, gethostbyname and bind use different types to represent IP addresses (see Figure 4(a)). In order to determine dependences between events, types of arguments and return values have to be normalized, e.g., all elements representing IP addresses should have the same type. We describe such a type-normalization procedure. Using types of arguments and return values to events, we define dependencies between events. Finally, dependencies between events are used to define forward and backward slices.

**Events and traces.** An event is a triple $e = \langle pc, event-name, arg-list \rangle$, consisting of the program counter $pc$, name of the event $event-name$, and a list of typed arguments $arg-list = (v_1 : \tau_1, \ldots, v_k : \tau_k)$ (where value $v_i$ has type $\tau_i$). An event represents a Win32 API call made at a specific location by the application. By convention, the last value in the argument list is the value returned by the event. A trace $T$ is simply a sequence $\langle e_1, \ldots, e_n \rangle$ of events.

**Type normalization.** The type mismatch problem is a result of the fact that equivalent data types can have different representations in C. For example, types `char*` and `char[]` are equivalent. Instead of using sophisticated type equivalence algorithms [10, 37] we normalize the types using the EDL specification. For example, the Windows’ socket API defines two structures that refer to hosts, shown in Figure 4(a). The structure `hostent` uses a `char**` to represent a list of possible IP addresses. On the other hand, the structure `sockaddr_in` stores a single IP address for a host using the type `in_addr` (which is a union of 4 bytes). Some calls like gethostbyname, and gethostbyaddr use `hostent` to return the address information for a given host. On the other hand, calls such as connect and bind use `sockaddr_in` to pass IP address information. Our slicing algorithm needs to deduce equivalence between types in order to determine dependency information.

Using the EDL specification we normalize all equivalent types and replace them by an unique type. This unique type is used in every place where one of the equivalent types is used in the API. Continuing with the example, we choose to use `addr_edl` as the unique type to represent IP addresses. We first create a unique type to represent the normalized type, and then update all corresponding data structures inside EDL to use this new data type. The corresponding EDL data structures (all the type names end with the suffix `edl`) are shown in Figure 4(b). Here the new unique type `addr_edl` is used to represent IP addresses in the two data structures `sockaddr_in_edl` and `hostent_edl`. This solution works as long as the types

![Figure 4: Using type normalization to improve slicing precision.](image-url)
are compatible, and the fact that both structures now use the same subtype allows the slicing algorithm to infer data dependency.

**Killed and used sets.** Consider an event \( e = \langle pc, \text{event-name}, \text{arg-list} \rangle \), where \( \text{arg-list} = (v_1 : \tau_1, \cdots, v_k : \tau_k) \). We associate two sets of values, called the killed set and used set (denoted by \( \text{killed}(e) \) and \( \text{used}(e) \)), with an event \( e \). The killed and used set are defined as follows:

\[
\text{killed}(e) = \{ v_i \mid v_i \text{ is a pointer and is modified by the event} \} \\
\text{used}(e) = \{ v_i \mid v_i \text{ is used by the event } e \}
\]

Notice that the killed set only contains pointers. Whether an argument is modified or used by an event is inferred from the documentation of the Win32 API call corresponding to the event. Note that the value corresponding to the return value of an event is always in the killed set.

**Computing Dependencies.** In order to define data dependence formally, we need to precisely define whether a value \( v \) depends on another value \( v' \). In traditional dynamic-slicing algorithms [4, 30, 53, 57], one records all the loads and stores at all memory addresses. Therefore, dependency information is easy to infer from memory addresses. Since our logs have incomplete information, deciding whether a value is dependent on another is tricky. We define a predicate \( \text{depends}(v : \tau, v' : \tau') \), which returns 1 if \( v \) is dependent on \( v' \) and returns 0 otherwise. Our definition of \( \text{depends}(v : \tau, v' : \tau') \) uses type information associated with values.

In our type system, certain primitive types (such as \text{addr}_\text{edl}) are designated as types with equality. Intuitively, primitive types with equality represent an operating-system resource or an entity (such as a host), e.g., a value of type \text{addr}_\text{edl} abstractly represents the host with the specified IP address. Two values of primitive types can only be compared if the types are designated as types with equality. In other words, given two values \( v : \tau \) and \( v' : \tau \), we say that \( v = v' \) is true iff \( v = v' \) and \( \tau \) is a primitive type with equality.

A record type that has \( k \) fields of type \( \tau_1, \cdots, \tau_k \) is denoted by \( \text{record}(\tau_1, \cdots, \tau_k) \). A pointer to an object of type \( \tau \) has type \text{ref}\( \tau \). Suppose that we have two values \( v \) and \( v' \) of type \( \tau = \text{record}(\tau_1, \cdots, \tau_k) \) and \text{ref}\( \tau \), respectively, where for \( 1 \leq i \leq k \), \( \tau_i \) are primitive types (\( v \) is a record and \( v' \) is a pointer to a record). Let the \( i \)-th field of the record \( v \) be \( v_i \) and the \( i \)-th field of the record pointed to by \( v' \) be \( (\star v')_i \). In this case, we say that \( v \) depends on \( v' \) if there exists \( i \) such that \( v_i = (\star v')_i \) and \( \tau_i \) is a primitive type with equality. Intuitively, the records represented by \( v \) and pointed to be \( v' \) share an operating system resource or entity corresponding to the primitive type \( \tau_i \). This intuitive idea is formalized and extended to the complete type system shown in appendix A.

Consider a trace \( T = (e_1, \cdots, e_n) \). A value \( v \) in an event \( e \) is denoted by \( (v, e) \). We say that there is a data dependence of value \( (v_i, e_a) \) on a value \( (v_j, e_b) \) if following three conditions are satisfied:

- \( v_i \in \text{used}(e_a) \),
- for all \( k \) such that \( b < k < a \) we have that \( v_j \notin \text{killed}(e_k) \), and
- value \( v_i \) depends on \( v_j \) or \( \text{depends}(v_i, v_j) = 1 \).

Notice that in traditional program analysis literature this is called a def-clear path from \((v_i, e_a)\) to \((v_i, e_a)\).

**Forward and backward slices.** Consider a trace \( T = (e_1, \cdots, e_n) \). The data-dependency graph or DDG of \( T \) is defined as a directed graph \( G = (V, E) \), where \( V \) and \( E \) are defined as follows:

- Consider an event \( e = \langle pc, \text{event-name}, (v_1 : \tau_1, \cdots, v_k : \tau_k) \rangle \). The set of vertices \( V(e) \) associated with \( e \) is \( \{ (v_1, e), \cdots, (v_k, e) \} \). The set of vertices \( V \) of the DDG is \( \bigcup_{i=1}^{n} V(e_i) \), where \( e_1, \cdots, e_n \) is the set of events occurring in \( T \).
- There is an edge from \((v, e_i)\) to \((v', e_k)\) (denoted as \( (v, e_i) \rightarrow (v', e_k) \)) iff \( v' \) depends on \( v \).
A forward slice from a value \((v_j, e_i)\) in the trace \(T\) (denoted by \(fslice((v_j, e_i), T)\)) is the sub-graph of \(G_T\) that contains all vertices that correspond to events which are reachable from \((v_j, e_i)\). A backward slice from a value \((v_j, e_i)\) in the trace \(T\) (denoted by \(bslice((v_j, e_i), T)\)) is the sub-graph of \(G_T\) that contains all vertices that can reach \((v_j, e_i)\).

### Implementation details and GUI.

Our dynamic-slicing tool is about 9000 lines of Java code. Description of the GUI and additional implementation details can be found in appendix B.2.

## 5 SandBoX86

The architecture of our infrastructure is shown in Figure 5. We first describe the life-cycle of an event through our architecture. Since our architecture monitors Windows applications, the events of interest are Win32 API calls. The number of the item given below corresponds to the arrows in Figure 5.

1. The application generates an event which is captured by the event-interceptor DLL.
2. The event generated by the application is transformed into an abstract event by the event-interceptor DLL and is passed to the policy-enforcer DLL.
3. The policy-enforcer DLL processes the abstract event and decides on an appropriate action.
4. If the abstract event is allowed, then the policy-enforcer DLL passes the concrete event to the operating system.
5. Results from the operating system are received and passed back to the application and the state of the monitor is updated. If the abstract event is denied, a failure code is returned to the application.

### Event Description Language (EDL).

The event description language or EDL is a platform-independent language for specifying parameterized events. It has a rich type system (described in appendix A), which allows for detailed specification of events and their associated arguments and return values. Intuitively, EDL describes abstract events which are used by the security policy. The intent of the EDL is that it can be used to create an abstract model for a given platform. For example, a model describing the Win32 and socket APIs would be an expected use of the EDL. We have implemented a compiler for EDL, which takes as its input a file with an EDL specification and produces header files which are used in generating the policy-enforcer DLL. We will describe various features of EDL using an example.

**Example 3** Consider the EDL example shown in Figure 6. The example defines the following data-types and events:

```plaintext
1. typedef sockaddr_edl = union string | int
2. typedef socket_edl = int
3. event connect(socket_edl sock, sockaddr_edl addr) = int
4. event gethostbyname(string name) = sockaddr_edl
```
- typedef sockaddr_edl... defines a type representing an IP address. It is a union because an IP address may be stored as either a string or an integer value.
- typedef socket_edl... defines a type representing a socket descriptor, stored as an integer.
- event connect(...)= int defines an event representing a connect system call. It takes two parameters, a socket, and an address. The event returns an integer error code.
- event gethostbyname(...) = sockaddr_edl defines an event representing a DNS nameservice lookup. It takes a hostname and returns an IP address.

**Event Sequence Language (ESL).** Our security policy is expressed in a language called the *event sequence language* or ESL. Examples of two ESL policies were given in Section 2. Therefore, due to space limitations we will keep the discussion of ESL brief. An ESL specification defines patterns on abstract events specified by the EDL. Once a pattern is matched, rules can be applied to determine whether or not the abstract event should be allowed or denied. Formally, ESL is a framework to describe a security automaton whose alphabet is the set of abstract events defined using EDL. We have built a compiler for ESL which takes files containing EDL and ESL specifications and creates the policy-enforcer DLL. We describe various components of ESL.

**Abstract events.** In Figure 3 of Section 2 [edl "kazaa.edl"] refers to the EDL file named "kazaa.edl". All rules defined in this ESL specification use the abstract events defined in the EDL file kazaa.edl.

**State.** The state in the security policy is defined using auxiliary variables. Any valid type in the C programming language is an allowable type for an auxiliary variable.

**Actions.** We define three types of actions in our security policy: allow, deny, and transform. The allow action specifies that an event should be allowed and passed on to the operating system. The deny action indicates that an abstract event should be disallowed and a failure code should be returned to the application. The transform action indicates how certain arguments of an event should be transformed before the event is passed to the operating system (OS). The transform action was added because certain applications will shut themselves down if system calls they issue fail. We are investigating OS-virtualization techniques to handle this issue.

**Rules.** Each rule in the security policy is of the following form:

```
match event [precondition] → action [postcondition]
```

An event \( e \) matches the above rule if the state of the security policy satisfies the precondition of the rule and \( e \) matches the event specified by the rule. If event \( e \) matches a rule, the action specified in the rule is taken and the state is updated according to the postcondition. If there are multiple rules, then the first rule that matches an event is applied (this is similar to processing of firewall rules). Events that do not match any rule are allowed by default.

**Implementation details.** The ESL and EDL compiler are about 2700 lines of Java code. For efficiency reasons, the monitor was implemented in C. Our implementation for the monitor is approximately 3000 lines of C code. Additional details can be found in appendix B.1.

5.1 Threats

A thorough discussion of traps and pitfalls of sandboxing systems which rely on system call interposition is given by Garfinkel [16]. The three major threats specific to our sandboxing system are:

**DLL bypassing.** A hostile application could try to thwart the interposed monitor DLL by not making calls through the standard Win32 API. We have implemented a kernel hook to address this attack. Our kernel hook is described in detail in Appendix C. Eventually, in order to eliminate the DLL bypassing attack we
want to move to a hybrid-sandboxing architecture\(^2\) which is a difficult task given the arcane nature of the Windows kernel API.

*Direct attack.* This attack can take two forms: modifying the monitor on disk or in memory with the goal of bypassing the security policy.

*Mimicry attack.* The idea behind this attack is to analyze the specific ESL policies in use, and transform the attack such that it is allowed by the security policy \(^{[55]}\). Countermeasures to direct and mimicry attacks are discussed in the appendix \(^{C}\).

## 6 Case Studies

To evaluate our infrastructure, we considered two applications: KaZaa \(^{[25]}\) and RealOne Player \(^{[41]}\). We chose these two applications because they have two properties in common: *both are very popular* and more importantly, *both are often classified as spyware*. As defined earlier, spyware is a useful software that also transmits private user data to an external entity. As both KaZaa and RealOne Player are used by millions of users, the prospect that they may contain malicious spying activities often causes great concerns. This section gives a brief overview of the two applications.

### 6.1 KaZaa

KaZaa \(^{[25]}\) is a distributed peer-to-peer (P2P) file-sharing application which allows millions of users to share files over the Internet. As of May 2004 \(^{[26]}\), an estimated 348 million copies of KaZaa had been downloaded. Although KaZaa itself may not have any spyware features, it often distributes third-party software with every release, and some, if not all, of these third-party software are spyware-laden. We divide these bundled spyware into three categories.

- **Advertising.** Most of the spyware distributed with KaZaa is related to advertising (and hence is called *AdWare*) and their main objective is to display advertisements to KaZaa users. Since KaZaa receives fees from third-party software vendors, there is an incentive for KaZaa to make sure that the users cannot bypass the AdWare. For example, Cydoor, a frequently cited spyware bundled with Kazaa, is in fact embedded within KaZaa so that a forceful removal of Cydoor will cause KaZaa to malfunction. The latest version of KaZaa that we have investigated (version 2.5.2) has the capability of thwarting users from using simple techniques to block its third party software. If KaZaa is unable to detect or undo the changes made by users, then it will crash when the built-in spyware attempts to connect to the home servers.

- **Spying.** Although KaZaa claims that it no longer bundles any spyware, older versions of software bundled with KaZaa had features such as tracking and reporting user activities back to the vendors and downloading and installing binary programs.

- **Hidden features.** When first revealed to the public in 2002, AltNet, a software bundled with KaZaa, caused an uproar amongst users. AltNet, once enabled, could allow AltNet’s distributor, Brilliant Digital, to tap into the vast computer resources of the KaZaa users. For instance, Brilliant Digital could use millions of KaZaa users’ computers to store and retrieve files, or to perform computation tasks. AltNet software is currently still bundled with KaZaa, even though Brilliant Digital has not activated it.

\(^{2}\)In a hybrid sandbox, the trapped event is first passed to a kernel module which consults a user module (which contains the security policy) to decide on the appropriate action for the event.
6.2 RealOne Player

RealOne Player is a part of the widely used software suite developed by Real Networks. RealOne Player can manage and play back many types of media files, such as real, mpeg and MP3. In addition, as a feature, RealOne Player can retrieve detailed information about the media file being played. For example, when a user is playing a music CD, RealOne Player can display the title of the CD as well as details about each track. Real Networks’ software has a well-known history of sending user-related information back to Real’s server, which is clearly stated in their End-User License Agreement (EULA). Although Real Networks does not reveal the details about what kind of information their software is collecting, it was widely reported that Real Jukebox from Real Networks used to send its user’s music preferences back to Real Networks’ home servers. For instance, when a user is listening to a CD, Real Jukebox could send information about the CD and the user’s unique identifier to Real Networks.

6.3 Thwarting Privacy Violations

Using a process very similar to the one described earlier (see scenario 2 in section 2) we were able to discover privacy violations in KaZaa and RealOne Player and gather detailed information about them. Using this information we were able to design security policies (which are very similar to the policy shown in Figure 3 of Section 2) to thwart these privacy violations. Details about this process are omitted due to space limitations.

7 Evaluation

This section describes an empirical study of the performance of sandBoX86. We evaluate cost over three sets of experiments. We measure low-level costs by collecting per-API call microbenchmarks, and profile application sandboxing using macrobenchmarks collected from user programs. Finally, we study the resource usage of our tool by analyzing memory consumption. These tests reveal that the performance and resource costs associated with sandBoX86 are nominal. Moreover, they further illustrate that these costs are less than those incurred by the majority of extant sandboxing tools, and did not demonstrably change the user experience in interactive applications (e.g., the sandboxing did not visibly change application response time). For example, the macrobenchmarks show a slow down of less than 3.2% for all three test applications.

We conducted all experiments on a 2.8GHz Intel Pentium-4 based machine with 1GB of memory, running Windows 2000 Professional. We use the high-performance timer (with the precision of 0.28µ-s/cycle) to obtain the experiment data. The SSH Secure Shell (version 3.2.0), KaZaa (version 2.5.2) and RealPlay (version 2.0) were profiled in our experiments. The SSH Secure Shell application was tested by connecting to a random host selected from a host pool. This test further simulates a SSH workload by connecting to the remote host, executing a fixed series of UNIX commands (e.g., cd, tar, gzip and grep), and disconnecting. The KaZaa application searched and downloaded the KaZaa installation binary (about 7MB), after which it exited. Finally, the RealPlay played a music file (about 3 minutes long) and terminated. Each experiment was conducted three times and results were averaged.

7.1 Microbenchmarks

Microbenchmarks measure the per-API call overhead. Our base metrics are obtained by measuring the time of each API call as uninstrumented (original in Table 1) and sandboxed. The difference between the two metrics is the measured total overhead of our sandbox infrastructure. We note that it is necessary to
Table 1: Microbenchmarks (partial)\(^a\)

<table>
<thead>
<tr>
<th>API Name</th>
<th>Application</th>
<th>Original ((\mu s))</th>
<th>Monitor</th>
<th>Overhead ((\mu s))</th>
<th>Enforcement</th>
<th>Logging</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind</td>
<td>RealPlay</td>
<td>78.78</td>
<td>5.87</td>
<td>1.68</td>
<td>16.34</td>
<td>23.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>66.99</td>
<td>5.92</td>
<td>1.57</td>
<td>16.17</td>
<td>23.66</td>
<td></td>
</tr>
<tr>
<td>closesocket</td>
<td>SSH</td>
<td>69.84</td>
<td>6.15</td>
<td>2.23</td>
<td>6.70</td>
<td>15.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RealPlay</td>
<td>64.81</td>
<td>6.29</td>
<td>1.54</td>
<td>8.66</td>
<td>16.49</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>57.93</td>
<td>6.31</td>
<td>1.47</td>
<td>7.93</td>
<td>15.71</td>
<td></td>
</tr>
<tr>
<td>connect</td>
<td>SSH</td>
<td>77.24</td>
<td>6.15</td>
<td>3.49</td>
<td>31.71</td>
<td>41.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RealPlay</td>
<td>126.41</td>
<td>6.29</td>
<td>4.75</td>
<td>20.11</td>
<td>31.15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>81.76</td>
<td>6.12</td>
<td>3.10</td>
<td>22.70</td>
<td>31.92</td>
<td></td>
</tr>
<tr>
<td>gethostbyname</td>
<td>SSH</td>
<td>31669.40</td>
<td>8.38</td>
<td>82.41</td>
<td>39.39</td>
<td>130.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RealPlay</td>
<td>7314.34</td>
<td>8.38</td>
<td>84.65</td>
<td>22.07</td>
<td>115.10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>34144.00</td>
<td>8.66</td>
<td>117.85</td>
<td>29.10</td>
<td>155.61</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>RealPlay</td>
<td>4736480.00(^b)</td>
<td>4.90</td>
<td>0(^c)</td>
<td>0(^c)</td>
<td>4.90</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>107134.30(^b)</td>
<td>4.20</td>
<td>0(^c)</td>
<td>0(^c)</td>
<td>4.20</td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>SSH</td>
<td>29.49</td>
<td>6.99</td>
<td>1.79</td>
<td>60.38</td>
<td>69.16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RealPlay</td>
<td>13.13</td>
<td>6.70</td>
<td>1.68</td>
<td>100.85</td>
<td>109.23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>38.13</td>
<td>5.95</td>
<td>1.56</td>
<td>45.25</td>
<td>52.76</td>
<td></td>
</tr>
<tr>
<td>socket</td>
<td>SSH</td>
<td>620.75</td>
<td>6.98</td>
<td>3.91</td>
<td>9.50</td>
<td>20.39</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RealPlay</td>
<td>286.53</td>
<td>7.26</td>
<td>4.84</td>
<td>9.08</td>
<td>21.18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>78.05</td>
<td>6.04</td>
<td>1.66</td>
<td>8.72</td>
<td>16.42</td>
<td></td>
</tr>
<tr>
<td>WSAAsyncSelect</td>
<td>SSH</td>
<td>8.89</td>
<td>5.81</td>
<td>1.44</td>
<td>8.52</td>
<td>15.77</td>
<td></td>
</tr>
<tr>
<td>WSARecv</td>
<td>SSH</td>
<td>10.48</td>
<td>5.88</td>
<td>1.46</td>
<td>12.67</td>
<td>20.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaZaa</td>
<td>40014.94</td>
<td>5.90</td>
<td>1.52</td>
<td>14.25</td>
<td>21.67</td>
<td></td>
</tr>
<tr>
<td>WSARecvFrom</td>
<td>KaZaa</td>
<td>11.56</td>
<td>5.94</td>
<td>1.50</td>
<td>23.94</td>
<td>31.38</td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)Due to space limitation, we only show a subset of the data. Trivial API calls such as \texttt{ntohl}, \texttt{htonl}, etc are omitted here.
\(^b\)The cost here includes the time spent waiting for I/O to complete, therefore is not an accurate reflection of the CPU overhead.
\(^c\)In this case, the policy does not enforce or log the \texttt{select} call. Therefore there is no overhead.

construct finer-grained metrics to capture the impact of application specific behavior. For example, policy enforcement is naturally application dependent. Additionally, because logging records the arguments of each API call, its cost could differ greatly from application to application (and in some cases, within the same application). For these reasons, we broke down the total overhead into three cost categories: monitor overhead, enforcement overhead and logging overhead. The results are shown in Table\(^\[1\]\).

### 7.2 Memory Usage

Note that in all tests, the \texttt{select} call appears to be abnormally expensive. This is because the measured \texttt{select} calls are blocking, and much of the measured time was actually spent waiting for the I/O to be ready, i.e., the program had yielded the processor. The actual CPU time for a \texttt{select} call should be a tiny fraction of the cost shown in the table. The measured cost of the \texttt{WSARecv} calls in the KaZaa application is also high for the same reason. Similarly, \texttt{gethostbyname} shows high execution cost in all three test cases also due to the blocking nature of the call.

The monitoring costs are small and nearly identical in all tests. This is not surprising because monitor-
7.3 Macrobenchmarks

While microbenchmarks gives a clear picture on the per-API overhead, it is the performance of the applications which end users care most about. For this reason, we measure the total overhead incurred by each application using macrobenchmarks. Shown in Table 7, these metrics reflect the aggregate cost by measuring the applications’ total run time. Out of the three applications tested, SSH incurred the highest overhead. For monitoring only, SSH took an additional 12.17 milliseconds of overhead; the total overhead for SSH is 27.54 milliseconds, or about 3.2% comparing to the original cost. In comparison, both RealOne Player and KaZaa only incurred a small amount of overhead (about 0.05% and 0.80%, respectively). In summary, the experiments indicate that interactive or computation bound applications will see little performance impact. Network bound applications may see some performance degradation, but even those costs are within reasonable limits (e.g., 3.2%).

Table 7: Macrobenchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>Orig. (ms)</th>
<th>Monitor</th>
<th>Overhead (ms)</th>
<th>Log</th>
<th>Total(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH Client</td>
<td>838</td>
<td>12.17</td>
<td>2.65</td>
<td>12.72</td>
<td>27.54(3.20%)</td>
</tr>
<tr>
<td>RealPlay</td>
<td>2146</td>
<td>0.43</td>
<td>0.14</td>
<td>0.43</td>
<td>1.00(0.05%)</td>
</tr>
<tr>
<td>KaZaa</td>
<td>4124</td>
<td>22.19</td>
<td>1.70</td>
<td>9.09</td>
<td>32.98(0.80%)</td>
</tr>
</tbody>
</table>

Table 8: Memory Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Original (KB)</th>
<th>Overhead (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Text</td>
<td>Data</td>
</tr>
<tr>
<td>SSH Client</td>
<td>2126</td>
<td>991</td>
</tr>
<tr>
<td>RealPlay</td>
<td>37</td>
<td>164</td>
</tr>
<tr>
<td>KaZaa</td>
<td>1143</td>
<td>1126</td>
</tr>
</tbody>
</table>
7.4 Memory Usage

In addition to the run-time overhead, we also measured the memory overhead, shown in Table 8. Since we use the same code-base for all three test cases, the text-memory overhead is the same for all applications. The data-memory overhead varies slightly among applications because of the various data structures we maintain are policy specific. Some data structures are shared among applications for efficiency reasons. For example, we maintain a cache of all function pointers to the libraries we are intercepting. This cache is shared among all monitored applications.

8 Conclusion

We presented the design and implementation of a sandboxing infrastructure for Windows. We also described a dynamic-slicing algorithm for analyzing data dependencies in event traces. Our dynamic-slicing algorithm can be used to discover privacy violations. Finally, we performed an evaluation of our techniques on two applications: KaZaa and RealOne Player.

There are several directions for future work. A sandboxing system is only as good as the security policy it enforces. We want to investigate techniques, such as model checking, for analyzing security policies. We also want to develop a more powerful query interface to our dynamic-slicing tool. We believe that this will improve the task of discovering privacy violations. We also plan to undertake a thorough study of weaknesses and vulnerabilities of sandboxing systems for Windows.

References


A The EDL type system

This section presents the type system used by EDL. We also present a formal definition of dependency between two typed values. A type $\tau$ is valid EDL type if it satisfies one of the following conditions:

- **Primitive types.** $\tau$ is valid if it is int, string, float, or one of the EDL-defined types, such as sin_addr_edl (which represents an IP-address).
- **Pointer types.** If $\tau$ is a valid type, then $\text{ref } \tau$ is a valid type ($\text{ref } \tau$ represents a a pointer to an object of type $\tau$).
- **Record and union types.** If $\tau_1, \ldots, \tau_k$ are valid types, then $\text{record}(\tau_1, \ldots, \tau_k)$ and $\text{union}(\tau_1, \ldots, \tau_k)$ are valid types. A record with $k$ fields of types $\tau_1, \ldots, \tau_k$ has type $\text{record}(\tau_1, \ldots, \tau_k)$.
- **Array type.** If $\tau$ is a valid type, then $\text{array}[\tau]$ is a valid type (which represents an array whose elements have type $\tau$).

Given a pointer $v : \text{ref } \tau_1$, $\text{deref}(v)$ has type $\tau_1$ and is the object pointed to by $v$, or $\text{deref}$ represents the dereferencing operator.

In the EDL type system we also define subtyping between primitive types. Given two primitive types $\tau_1$ and $\tau_2$, $\tau_1 \leq \tau_2$ denotes that $\tau_1$ is a subtype of $\tau_2$. The subtyping relationship $\leq$ between primitive types can be extended to the entire EDL type system in a standard manner [10].

A.1 Defining the predicate $\text{depends}(v : \tau, v' : \tau')$

Recall that $\text{depends}(v : \tau, v' : \tau')$ returns 1 if $v$ is dependent on $v'$ and returns 0 otherwise. First, we define an operator called $\text{flatten}(v : \tau)$ which returns a list of values of primitive and pointer types contained within $v$. Intuitively, the $\text{flatten}(\ldots)$ operator recursively flattens all records to create a list of all values with primitive and pointer types occurring in it. We do not consider union types because the actual type of a value is known at runtime. Similarly, array bounds are also known at runtime.

- If $\tau$ is a primitive or pointer type, then $\text{flatten}(v : \tau)$ returns $(v : \tau)$.
- If $\tau$ is of type $\text{record}(v_1 : \tau_1, \ldots, v_k : \tau_k)$, then $\text{flatten}(v : \tau)$ is the concatenation of the lists $l_1, \ldots, l_k$ where $l_i = \text{flatten}(v_i ; \tau_i)$.
- If $\tau$ is of type $\text{array}[n] \tau_1$, then $\text{flatten}(v : \tau)$ is the enumeration $\text{flatten}(v[1]), \ldots, \text{flatten}(v[n])$ of the array.

Suppose we are given two lists $l = (v_1 : \tau_1, \ldots, v_m : \tau_m)$ and $l' = (v'_1 : \tau'_1, \ldots, v'_n : \tau'_n)$ of values of primitive and pointer types contained within $v$. We say that the intersection of $l$ and $l'$ is non-empty if there exists two values $v_i : \tau_i$ and $v_j : \tau_j$ in $l$ and $l'$ respectively such that $v_i = v_j$ and one of the following two conditions is true:

- $\tau_i$ and $\tau_j$ are pointer types.
- $\tau_i = \tau_j$ and $\tau_i$ is a primitive type with equality.

The predicate $\text{depends}(v : \tau, v' : \tau')$ is defined to be 1 if the following condition is true (in all other cases it is defined to be 0):

Type $\tau'$ of $v'$ is a pointer (or of type $\text{ref } \tau_1$) and the intersection of the two lists $\text{flatten}(v : \tau)$ and $\text{flatten}(\text{deref}(v') : \tau_1)$ is non empty.
B Implementation Details

B.1 SandBoX86

Implementation of ESL/EDL compilers. The implementation of the compilers is about 2700 lines of Java. The EDL compiler transforms type declarations into intermediate type-description objects, and event declarations into event declaration objects. From these intermediate data structures, appropriate C headers and C logging functions are generated. The ESL compiler first invokes the EDL compiler to parse the referred-to EDL. Subsequently, the ESL compiler processes the variable and event policy definitions into its own intermediate data structures. Where appropriate, these are type-checked against the corresponding EDL types. After the structures are built, the ESL output phase coalesces all match statements for a given event into a single C policy function that tests the event preconditions in order and returns the matching event policy. The collection of these C functions is the output of the EDL compiler, which is then compiled into the policy-enforcer DLL.

Implementation of the Monitor. Recall that the monitor consists of the event-interceptor and policy-enforcer DLL. The event-interceptor DLL consists of wrappers for Win32 API compatible functions. The event-interceptor DLL is inserted into the application’s execution search path where it overrides the system DLL of the same name. Calls to API functions in the DLL are therefore trapped by the event-interceptor DLL, which passes the corresponding abstract event to the monitor. The monitor logs the event and determines the appropriate action by using the policy-enforcer DLL. There are about 3000 lines of C code that make up the monitor, of which about 1000 lines are generated by the EDL compiler.

B.2 Dynamic Slicing Tool

While monitoring a program, execution traces are created using logging functions, which are auto-generated by the EDL compiler. These logging functions are directly generated from the EDL, which are constructed to model the operating system events which are being monitored. The logging functions (generated by the EDL compiler) record events with the corresponding parameters. Stack walking techniques are also used to log the additional program state, such as the instruction pointer. When the program executes with the monitor in place but without a security policy, the execution trace will be logged to a file. This file is then processed by our dynamic-slicing tool. Our dynamic-slicing tool has a GUI-frontend which can be used to pose slicing queries. Our dynamic-slicing tool is about 9000 lines of Java code.

We have also implemented a GUI frontend to our dynamic slicer. The GUI allows a security analyst to specify queries, perform forward and backward slices, and browse the slices.

C Threat Model

C.1 DLL bypassing

A hostile application could try to thwart the interposed monitor DLL by not making calls through the standard Win32 API. For example, by statically linking the DLLs or making direct kernel calls.

Analysis: Although this may work in the short term, an application that does this is not likely to be very portable across different releases, or even service-pack levels of Windows. The expense of developing and
maintaining these solutions is high, so we do not consider this to be a widespread attack, however, the possibility remains.  

**Countermeasures:** There are two approaches to address this attack. The first solution is to monitor events at the the kernel level. However, this is equally as difficult for the anti-spyware authors because the Windows kernel API is low-level, poorly documented, and subject to change. In short, it is not a level of abstraction that lends itself to interposition. A more practical solution is to install a simple kernel monitor that detects attempts to bypass the standard API and shuts them down. We have implemented such a kernel monitor.

Our implementation of the kernel monitor is based on the *strace* program for Windows [40]. The interposed DLL communicates with the kernel monitor using a “special” native Win32 API call. The requirements for this special call are as follows: should be rarely used, should accept multiple arguments, and should not have harmful side effects if the kernel monitor is not loaded. We chose the `NtPrivilegedServiceAuditAlarm` call, which satisfies all the requirements mentioned above. This call takes 5 parameters. When the DLL uses this call to communicate with the monitor, 4 parameters are set to specific random values. This allows the kernel monitor to recognize that the call is intended for itself, and not for the Windows kernel. In addition, the random values provide a level of protection from attempts to bypass this mechanism. The random values are generated on each system on which the monitor is installed on, and a malicious program would have to perform on-the-fly reverse engineering of the DLL to discover the random values. The fifth parameter is used as a flag which enables or disable events.

When a Win32 call is intercepted by the interposed DLL, it first issues the `NtPrivilegedServiceAuditAlarm` with enable flag, which indicates to the kernel monitor to enable system calls. Then the actual Win32 call is processed. Finally, before returning control to the application, the `NtPrivilegedServiceAuditAlarm` with enable flag is issued, which indicates to the kernel monitor to disable system calls.

### C.2 Direct attack on the monitor

This attack can take two forms: modifying the monitor on disk or in memory with the goal of bypassing the security policy.

Figure 10: Trace Analyzer GUI. The left panel shows the trace files being analyzed. The top-right panel allows user to analyze the trace and results are shown in the tabular format at the bottom-right panel.
Analysis: We believe that if a significant number of users deploy a sandboxing technology such as sand-BoX86 to thwart spyware, the above-mentioned attack will be the most likely technique used by the spyware developers. There is relatively little expense or downside to implementing this attack.

Countermeasures: The on-disk attack can be thwarted simply by applying sensible and routine file security at the operating-system level. The untrusted application should not be run at a security level that allows it to write to arbitrary files and directories. This works well if, for example, sandBoX86 is installed by a system administrator. If for some reason, the monitor and application run at the same security level (for example, if they are both installed by an untrusted user of the system), the ESL policy can be written to include self-protection measures against this attack. The in-memory attack can be thwarted by using CPU features to write protect the memory region where the monitor is loaded, and then disallowing the application to make operating system calls that would undo the protection. The binary code for the monitor can also be obfuscated to make reverse engineering hard.

C.3 Mimicry attack

The idea behind this attack is to analyze the specific ESL policies in use, and transform the attack such that it is allowed by the security policy. Such attacks on host-based intrusion detection systems, called mimicry attacks, are discussed by Wagner and Soto [55].

Analysis: As anti-spyware software becomes more sophisticated, the spyware developers will take more and more steps to disguise their actions. This is a common problem in security. Unlike the previous attacks which try to thwart the monitor at a systems level, and thereby have systems level solutions, this attack works within the security policy. In theory, a "perfect" security policy would allow every desired behavior and disallow every undesired behavior.

Countermeasures: The best solution to this is to provide tools to allow the "good guys" to stay ahead in this arms race. Modeling tools that allow "what-if" scenarios to be evaluated in the context of a proposed security policy are helpful to examine the consequences of different behavior patterns the application might try. The slicing and analysis infrastructure is an invaluable tool for determining what behaviors need to be addressed in a given version of an application, of course, this analysis needs to be kept current when new software is deployed.