Combating Privacy Violations in COTS Software *

Research Proposal for Ph.D. Preliminary Exam

Hao Wang
University of Wisconsin-Madison

September 20, 2004

1 Introduction

Protecting privacy of information is an important goal of computer security. Information can refer to either tangible resources, such as files stored on a computer, or intangible events, such as the action of playing a music CD or logging on to a website. Software can violate privacy in one of two ways: *through unauthorized access of resources* and *by leaking sensitive information via unauthorized channels*, or both. We refer to the first type, *unauthorized access of resources*, as Type I privacy violation, and the second type, *information leakage*, as Type II privacy violation. To combat privacy violations in software, therefore, is to detect and stop these two types of violations.

Although the importance of protecting privacy has long been recognized, and much research has been done in this area, existing operating systems do not provide sufficient support for detecting and preventing privacy violations caused by *software running inside the systems*. In most modern operating systems, such as Linux and Windows, the main security objective has been protecting the integrity of the systems from outside attacks. This means protecting the kernel from users; and protecting users from each other. Hence, access rights to resources are granted at the granularity level of individual users, and every software run by a user has the same set of privileges as the user, regardless of what the software actually does. This lack of fine-grained access control provides opportunities for malicious software to abuse the bestowed trust and violate users's privacy by stealing information from the unsuspecting users. This problem, illustrated in Figure 1, has long been exploited by malicious software such as *trojans*, *key-loggers*, and, more recently, *spyware* [38].

There are two general approaches for combating privacy violations in software, mirroring the two types of violations discussed above. One approach, aimed at preventing Type I privacy violations, is called *discretionary access control*. The other approach, addressing Type II privacy violations, is called *information flow control*, or *mandatory access control*.

Discretionary access control-based techniques protect privacy by controlling access to private resources, as illustrated in Figure 1 (a). This is usually done through run-time mechanisms such as sandboxes or reference monitors. Consequently, these techniques generally work with binaries and do not require source code.

One major drawback of discretionary access control-based techniques is that they do not prevent unauthorized information flow. Once software is granted access to a piece of information, it can use the infor-

*COTS stands for Commercial Off-The-Shelf.

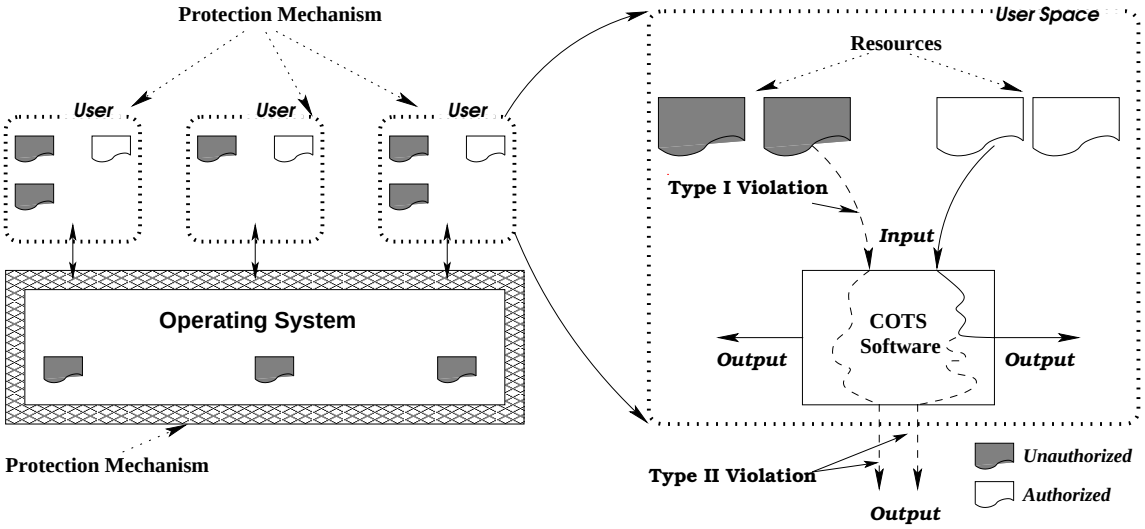


Figure 1: Type I and Type II Privacy Violations.

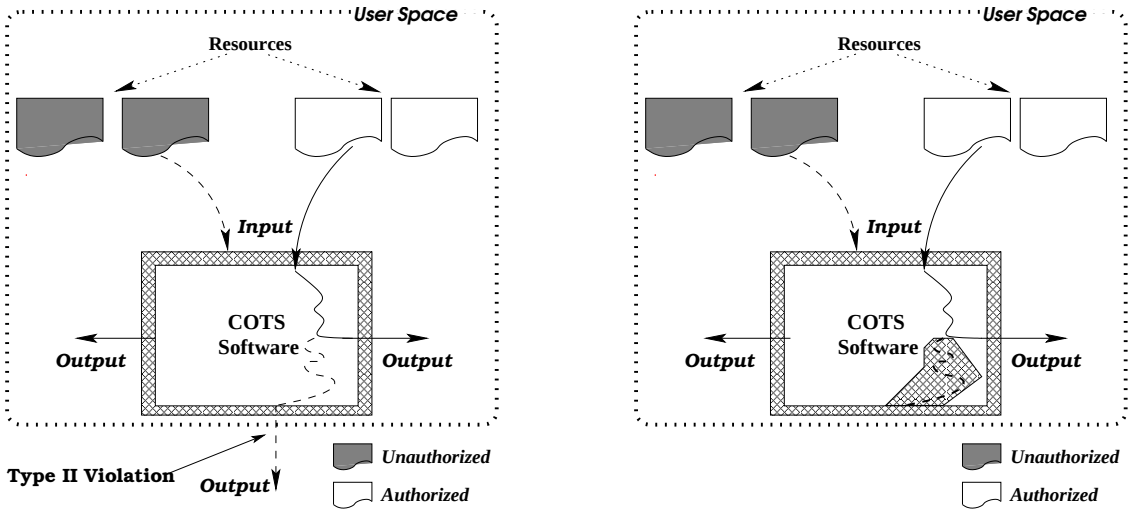
mation in anyway it wishes. For instance, the software can send the information to unauthorized entities, therefore violating a user's privacy. Information flow control-based techniques address this shortcoming by tracking and controlling dataflow inside the software and therefore can prevent privacy violations caused by unauthorized flow of information. Figure 1 (b) demonstrates this type of approaches.

Traditional information flow control-based techniques were developed under the assumption that source code was available for analysis. This assumption came primarily from military environment where software used to be developed in-house. Therefore source code was available and could be analyzed using static analysis techniques to detecting potential privacy violations. In non-military environment, such as commercial companies and consumer markets, this assumption does not hold. *Commercial Off-The-Shelf (COTS)* software, which is usually distributed in binary format only, is used instead. Furthermore, for economic reasons, today's military systems have begun to rely on COTS software too [31, 8].

COTS software poses new challenges for both *discretionary access control* and *information flow control* which existing work in these areas does not address.

1. **Principle of Least Privilege:** The *principle of least privilege* states that *a principal should receive the minimum access right required to execute a task* [44]. Traditional discretionary access control approach, however, grants access rights based on principals (i.e. users), not on tasks. Software, therefore, has the full set of privileges as the user, regardless of what it actually needs. For a traditional system where software is developed in-house from scratch, this approach may work well since the software usually goes through auditing and testing, and can therefore be trusted.

   However, COTS software comes from various sources, including both reputable vendors as well as dubious distributors. COTS software, therefore, may abuse the extra privileges granted by discretionary access control and violate a user's privacy. Consequently, it is necessary to grant access right to COTS software based on the tasks it performs. Without understanding the behavior and resource requirements of COTS software, discretionary access control may be either too restrictive or too lax, and neither is desirable. New techniques must be developed to capture and analyze run-time behaviors of binary software such as COTS to meet the principle of least privilege.



(a) Addressing Type I privacy violations. Notice that Type II privacy violations may still occur.

(b) Addressing Type II privacy violations.

Figure 2: Addressing Type I and Type II privacy violations.

2. **Information flow control for COTS:** Traditionally information flow control focuses on detecting privacy violations in source code, using static analysis techniques. This approach does not work for COTS software since in most cases COTS software is only distributed in binary format. However, analyzing binaries using static information flow techniques is a difficult task, since type information is usually stripped from the binaries. In addition, static analysis techniques cannot capture run-time parameters (such as environment variables and input arguments) that may affect dataflow. Therefore static analysis techniques for addressing information flow have to be conservative, and hence not precise.

   An alternative approach to static information flow control is to analyzing and controlling information flow dynamically, at software run-time. Dynamic information flow analysis can capture run-time variables and therefore may obtain more precise results than static techniques. However, improved precision comes at the cost of performance: dynamic information flow analysis of binaries tends to have high run-time overhead and therefore is not practical. This is a challenge that must be met in order to provide efficient run-time information flow control for COTS.

Given these new challenges, I propose to study techniques for combating privacy violations in COTS software. My work centers around two basic questions:

1. *How to discover privacy violations in COTS software?*
2. *How to contain privacy violations in COTS software efficiently?*

Accordingly, my work makes following contributions towards combating privacy violations in COTS software.

- **Providing tools for analyzing and understanding COTS software:** First, I provide a set of tools for analyzing COTS software. The objective is to uncover potential privacy violations in COTS software. Furthermore, since real world COTS software is large and analyzing binaries is a tedious task, it is very important to provide end-users with a suite of automatic-analysis techniques.

- **Providing efficient mechanisms for controlling information flow in COTS software:** Another contribution of my work is to provide efficient run-time mechanisms for controlling information flow in COTS software. Existing work focuses on analyzing information flow in source code, not binaries such as COTS software. My work will address this gap.

- **Dealing with real-world COTS software:** I want to not only developing *theoretical* knowledge towards preventing privacy violations in COTS software, but also providing *practical* implementations for addressing the problem. I will develop tools and techniques that can be used on real-world COTS applications.

2 Related Work

Protecting the privacy [1] of information, or simply *privacy protection*, has always been an important goal of computer security. Lampson first summarized this problem, and called it the *confinement problem* [36]. Lampson noted that privacy protection not only means preventing unauthorized access and manipulation of information, but also requires detecting and blocking information leakage–often caused by *authorized processes*. Preventing unauthorized access is the issue of *discretionary access control* and has been well studied [35, 22, 49, 26, 4, 43, 15]. Detecting information leakage is a much harder problem and requires *information flow analysis*. Previous work in combating privacy violations can be divided into these two categories: those that use discretionary access control to prevent unauthorized access, and those that rely on information flow control to stop information leakage.

Although much work has been done in both *discretionary access control*- and *information flow control*-based privacy protection, none of the previous work completely addresses the new challenges brought by Commercial Off-The-Shelf (COTS) software. Discretionary access control-based techniques do not take run-time behavior and resource requirements of each COTS software into consideration, and therefore violate the *principle of least privilege*. Most information flow control-based techniques only operate on source code, not binaries–the common format COTS software distributed in. My work is to address these shortcomings in existing privacy protection techniques.

2.1 Discretionary Access Control

2.1.1 Classical Work on Discretionary Access Control

Classical work in discretionary access control was based on the *access matrix* model, which was first described by Lampson [35] and later extended by Denning [22] and Graham [27]. This model contains a set of *subjects*, a set of *objects* and a set of *permissions*. The access matrix is essentially a table with one subject for each row and one object for each column. Each cell in the matrix contains the permissions granted to the subject for the corresponding object. In most systems, the subjects are users and the objects are the protected resources. Software run by a user naturally inherits all the permissions granted to the user.

The matrix model and its derivatives are very popular for their simplicity and appear widely in many systems. However, this approach does not strictly follow the *principle of least privilege*, which states that

[1] Privacy is also called confidentiality in existing security literature. In the rest of this document, I use them synonymously.

a principal should receive the minimum access right required to execute a task. The access matrix model, instead, grants permissions based on *principals (users in this case)*, not on the *tasks being executed*. Consequently, software executes with all the privileges the user has—even if the software does not require them. This may not be an issue for traditional systems where software is often developed in-house, from scratch, and therefore source code is available for testing and analysis, including verification of security properties. However, in environment where Commercial Off-The-Shelf (COTS) software is used, access matrix model, and consequently classical discretionary access control, is not suitable for combating privacy violations. COTS software is normally distributed in binary format only, and it can come from various sources, including both reputable commercial companies as well as vendors with malicious intentions. Classical discretionary control-based techniques do not have the expressiveness required to grant access rights based on what software does and what software needs, as required by the principle of least privilege.

2.1.2 Recent Work on Discretionary Access Control

One major drawback of the classical works on discretionary access control is that they lack the flexibility needed to grant customized access control on a per-software basis. This shortcoming is the fundamental reason for the *root shell* [30] attacks, which exploit software bugs in order to obtain privileged access to a system. This problem is well recognized by the security community and much of the recent work on discretionary access control aims at fixing this shortcoming.

Recent works [49, 26, 4, 43, 15] on discretionary access control try to offer more customizable and efficient access control by providing a policy-based framework. Users can express application-specific security policies which list the access privileges granted to an application. Different policies can be created for different software, depending on users' view of the software and security requirements.

Janus [49, 26] MAPbox [4], Systrace [43] as well as TRON [15] are examples of policy-based access control. Janus allows a customized, application-specific policy. MAPbox builds on Janus and therefore inherits, and extends its policy language. MAPbox differs from Janus in that it classifies applications into behavior classes and associates different security policies with each behavior class. This is a weaker version of the principle of least privilege since access rights are granted per group, not per software. Systrace uses a simple list based language to express its security policies for different applications. TRON extends the previous work by entirely de-coupling policy from application and allows arbitrary policy to be attached to any application at run time.

Although these policy-based discretionary access control approaches attempt to achieve the principle of least privilege, they fall short of the goal. These approaches only provide mechanisms for specifying and preventing privacy violations, but not tools for recognizing privacy violations. Instead heuristics are commonly used to construct the policies specifying the violations. These heuristics *do not* necessarily reflect the true nature of a software and therefore may either miss certain privacy violations (false negatives) or are too restrictive and therefore interfere with the correct operation of the software (false positives).

Another type of approach detects privacy violations at the time when COTS binaries are delivered and installed. Virus scanners (and malware scanners in general) [5] are examples of this approach. They detect malicious binaries by scanning files (of any format) using signatures of known malicious code. However, as Christodorescu and Jha have demonstrated in their work [19], these signature-based approaches can be easily defeated by using program transformation on the malicious code. Another drawback of this type of approaches is that it cannot detect malicious code not defined in the signature database.

In order to achieve the principal of least privilege, a better understanding of the real behavior and resource requirements of software is necessary. There has been very little work done in this area. Ammons et

al. propose a specification mining-based approach [10, 9]. They use machine learning techniques to analyze program call traces in order to capture formal specifications of the software. Their objective is to uncover program specifications, not malicious usage patterns. King and Chen propose BackTracker system which reconstructs adversarial behavior from traces of operating system events on compromised hosts [32]. BackTracker system uses timed file-system and process system call traces to reconstruct possible compromising events. However, their objective is understanding intrusion detection, not software analysis. Bergeron et al. also investigate how to analyze binaries to isolate malicious behaviors [14, 13]. They use slicing techniques to statically analyze a binary and produce a special graph (called critical API graph) representing the call patterns for the binary. The graph is compared with a manually created signature graph that records undesirable call patterns. If the binary's graph matches the signature graph, then the binary is said to contain malicious behavior. Their approach does not uncover new malicious behaviors, only the ones that are represented through the signatures. Therefore, their approach cannot detect potential privacy violations not recorded in the signature database.

Both classical and recent work on discretionary access control-based privacy protection share one inherent limitation: they cannot address *information flow leakage*. Under discretionary access control, authorized software can still leak protected information to unauthorized entities, usually through *covert channels*. Schneider [45, 46] provides a general model for describing this problem. He categorizes security policies based on underlining enforcement mechanisms and defines the set that can be enforced by monitoring execution traces of target programs the *Execution Monitoring (EM) enforcible policies*. Access control policies can be represented as *security automata* and therefore are EM enforceable. Janus, MAPbox, Systrace and TRON fall into the EM enforceable framework. However, information flow control-based approaches are not EM enforceable and therefore require other techniques.

2.2 Information Flow Control

The second part of the confinement problem, stated by Lampson, observes that even with access control in place, information leakage is still possible. This information leakage can occur either explicitly or implicitly and is the focus of work on information flow control. Previous work in this area can be divided into two areas: *static information flow control* and *dynamic information flow control*. Static information flow control only operates on source code, normally at compile time. Dynamic information flow control works with binaries at run-time.

2.3 Static Information Flow Control

Information flow control based work was initiated by Bell and LaPadula [12]. Their security model, the BLP model, uses a set of axioms to control information flow inside a system. The *no read-up* axiom states that a user can only access information at or below her security level. The *no write-up* axiom guarantees information cannot leak from high level objects to low level objects. The BLP model is applicable to both static and dynamic information flow control. Denning [20] used the lattice model to capture information flow within a system. The lattice model is a more formal representation of the BLP model and it provides a foundation for information flow analysis and control. Denning and Denning later proposed the idea of *Program certification* [21] based on the lattice model. Program certification is a static approach that uses language-based techniques to analyze source code for information flow violations at compilation time.

More recent work uses type systems as the basis to control information flow and verify program safety. Volpano et al. [48] provide a simple language which encodes Denning's lattice model. They apply language-based techniques to prove the security of programs written in this simple language. SLam calculus [29, 3] maintains and propagates both type and security information in a functional language (the λ-calculus). One

drawback of these language-based approaches is that they only work on simple language-based systems and are not practical at present time.

An exception to the type systems-based work is the Java Information Flow (JIF) [39] language extension and the JFlow run-time system developed by Myers and Liskov [39]. JIF is based on Java, a popular and practical programming language. JFlow system provides simple run-time checking for information flow. However, their work is limited to Java and therefore is not applicable to the vast majority of binary programs written using C or C++. In addition, their main focus is to address multi-level security in military style systems, *not to handle privacy violations in software*.

There is very little work on static analysis of binaries for checking information flow. This is mainly because high level type information normally is not present in binaries. COTS software is normally distributed in stripped binary format, without any symbol information. Therefore, preforming precise data flow analysis on binaries is a very hard problem, as demonstrated by the x86 binary analysis work from Balakrishnan and Reps [11]. Their work focuses on uncovering precise memory related information such as locations and access patterns. While the technique can be used to improve the precision of binary analysis, it is not yet applicable to large binaries.

In summary, static information control-based techniques focus on language- and type-system- based approaches and can be only applied to source code. In an environment where COTS software is used, these techniques do not work. Furthermore, analyzing binaries statically is a hard problem due to the lack of high level type and symbol information.

2.4 Dynamic Information Flow Control

It is also possible to enforce information flow control dynamically at run-time. Prior work in this area includes the Data Mark Machine proposed by Fenton [23, 24]. The Data Mark Machine associates a security class with the program counter (PC) of a process. This security class is changed dynamically via push and pop as the process executes instructions. Whenever a new instruction is to be executed, the old class is first pushed onto the stack, and the new security class information is associated with the PC. Once the instruction is executed, the old security class is restored. Gat and Sall extends the Data Mark idea to machines with general purpose registers, and propose that the *push* and *pop* operations should be applied to registers as well [25].

High run-time overhead is a major drawback of dynamic information flow control-based systems. As is obvious, with Fenton's Data Mark machine, a fair amount of computation must be done at each step. Not surprisingly, in the past very little work has been done on enforcing information flow at run-time. Instead, a great deal of work has been done on detecting information flow statically.

3 Discovering and Containing Type I Privacy Violations (Unauthorized Resource Usage)

Adhering to the principle of least privilege is the key to preventing Type I privacy violations (*unauthorized resource usage*) in COTS software. The fundamental problem in enforcing this principle is finding the right set of privileges for each application. An application may abuse the extraneous privileges it receives; conversely, if the application receives insufficient privilege, it may not be able to perform its tasks.

Existing work on combating privacy violations, however, may not satisfy the principle of least privilege, as access privileges are granted only according to security objectives, but the actual program behavior is not taken into consideration. This problem is captured in Figure 3. The success of this approach is entirely dependent on the correct implementation of the security objectives, represented as the security policy. The policy may be either too strict or too lax, and neither is desirable.

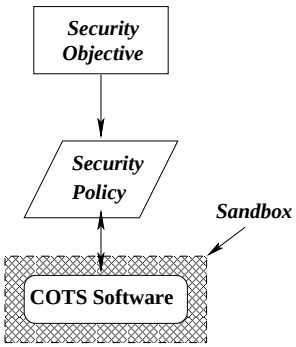


Figure 3: Traditional approaches for addressing privacy violations

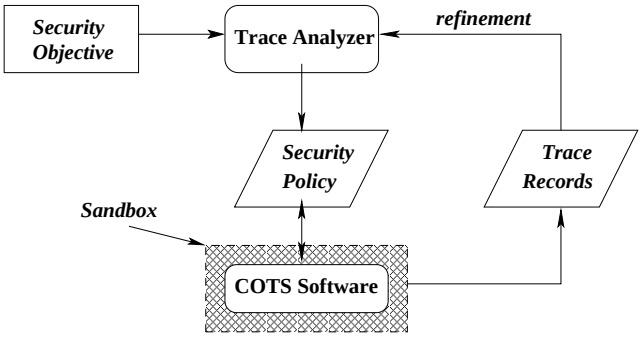


Figure 4: Our approach for addressing privacy violations

Our approach, shown in Figure 4, achieves the principle of least privilege by adding a *refinement* step, where the security policy can be refined by using information obtained from analyzing the runtime behavior of a program.

We illustrate our approach through a concrete example. Assume that a user wants to prevent an application from contacting certain servers and receiving pop-up ads from them. Furthermore, the servers may change frequently during the lifetime of the application. Clearly, the user has insufficient information to implement a policy that can restrict the application according to her security objective. Our approach remedies the situation by collecting runtime traces of the program, and then analyzing the collected traces to extract required information for constructing the security policy. Furthermore, the refinement step can be repeated so that dynamic changes, such as the variation of servers in this example, can be captured.

In this section we present our refinement process. We use a type system, called *event definition language (EDL)*, to abstract programs' runtime behavior into sequence of events. Security policies are constructed using the *event specification language (ESL)*. An ESL policy specifies patterns of undesirable behavior (e.g. privacy violations), using abstract EDL events, and the corresponding actions (e.g. allow or deny) to take when the patterns are matched. Our sandboxing tool intercepts a program's library calls and transforms them into abstract event sequences using the EDL. After event traces are collected, the trace analyzer is used to analyze the traces using a *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 traces.

3.1 Event Description Language (EDL)

The *event description language (EDL)* is a platform-independent language for specifying parameterized events. It has a rich type system (described in our technical report [34]), 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 an EDL specification and produces header files which are used in generating the policy-enforcer DLL.

We will describe various features of EDL using the example shown in Figure 5. This example defines the following data-types and events:

```
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
```

Figure 5: EDL example.

```
1. edl "kazaa.edl"
2. hash iptable
3. string blocked = "cydoor|doubleclick|adserver|fastclick.com|..."
4. match gethostbyname(blocked) → allow ≪addHash(iptable, ret, name)≫
5. match connect(...,host,...,≪checkHash(iptable, host)≫) → return success
6. match sendto(...,host,...,≪checkHash(iptable, host)≫) → return success
7. match recvfrom(...,host,...,≪checkHash(iptable, host)≫) → return success
```

Figure 6: An ESL policy example.

- **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.

3.2 Event Sequence Language (ESL)

Our security policy is expressed in a language called the *event sequence language* or *ESL*. 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 [45] 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, using an example ESL policy shown in Figure 6.

Abstract events. In Figure 6, [**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. In Figure 6, for example, `iptable` is a hash table used to keep track of IP addresses for the policy.

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). For example, the *remote host* argument of a `connect` call may be transformed into a localhost. 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]* $\rightarrow$ *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. For example, in Figure 6, line 4 of the policy states that any call to `gethostbyname` is allowed; if the hostname is in the block list, then the corresponding IP address is stored in the hash table.

3.3 Understanding Program Using Dynamic Slicing

The key to combating privacy violations is to uncover evidence of privacy-violating behaviors. In our approach, evidence is presented in the form of *execution traces*. An execution trace consists of multiple EDL events, where each event corresponds to a Win32 API call [2]. Dynamic slicing is used to discover dependencies between events in an execution trace. These dependencies can then be used to discover privacy violations and gather information to construct security policies.

Figure 7 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. The example shows the use of *backward slicing* to find all the hosts a program has sent data to using the `send` system call.

This section 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 8 (a)). In order to determine dependencies 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 $e = \langle pc, \textit{event-name}, \textit{arg-list} \rangle$, consisting of the program counter *pc*, name of the event *event-name*, and a list of typed arguments *arg-list* $= (v_1 : \tau_1, \cdots, v_k : \tau_k)$ (where value v_i has type τ_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 $\langle e_1, \cdots, e_n \rangle$ of events. We use the event description language (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.

[2] We currently target our tools at Windows. However, the techniques can be applied to other systems as well.

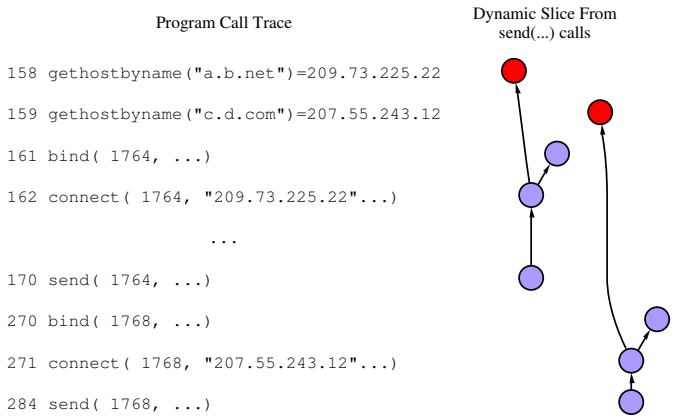
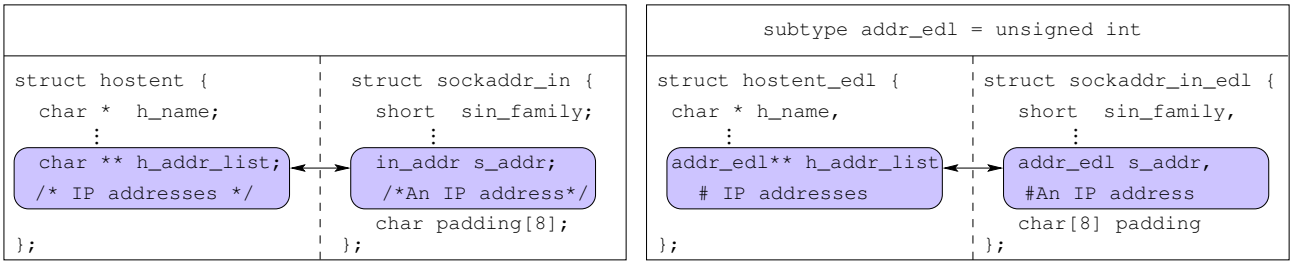


Figure 7: Information gathering using dynamic slicing.



(a) Two different ways to store IP addresses **(b) Using normalized data structure to store IP addresses**

Figure 8: Using type normalization to improve slicing precision.

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 [17, 42] we normalize the types using the EDL specification. For example, the Windows' socket API defines two structures that refer to hosts, shown in Figure 8 (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 `struct 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.

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 8 (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 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 $e = \langle pc, \textit{event-name}, \textit{arg-list}\rangle$, where *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 *killed(e)* and *used(e)*), with an event e. The killed and used set are defined as follows:

$$\begin{aligned} \textit{killed(e)} &= \{v_i \mid v_i \text{ is a pointer and is modified by the event}\} \\ \textit{used(e)} &= \{v_i \mid v_i \text{ is used by the event } e\} \end{aligned}$$

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 [6, 33, 47, 52], 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 *depends*$(v : \tau, v' : \tau')$, which returns 1 if v is dependent on v' and returns 0 otherwise. Our definition of *depends*$(v : \tau, v' : \tau')$ uses type information associated with values.

In our type system, certain primitive types (such as `addr_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 `addr_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 τ is a primitive type with equality.

A record type that has k fields of type $\tau_1, \cdots, \tau_k$ is denoted by $record(\tau_1, \cdots, \tau_k)$. A pointer to an object of type τ has type $ref\,\tau$. Suppose that we have two values v and v' of type $\tau = record(\tau_1, \cdots, \tau_k)$ and $ref\,\tau$, respectively, where for $1 \leq i \leq k$, τ_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 τ_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 τ_i. This intuitive idea is formalized and extended to the complete type system shown in our technical report [34].

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:

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

Notice that in traditional program analysis literature this is called a def-clear path from (v_j, e_b) to (v_i, e_a) [7].

Forward and backward slices. Consider a trace $T = \langle e_1, \cdots, e_n\rangle$. 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 $e = (pc, \textit{event\_name}, (v_1 : \tau_1, \cdots, v_k : \tau_k))$. 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.

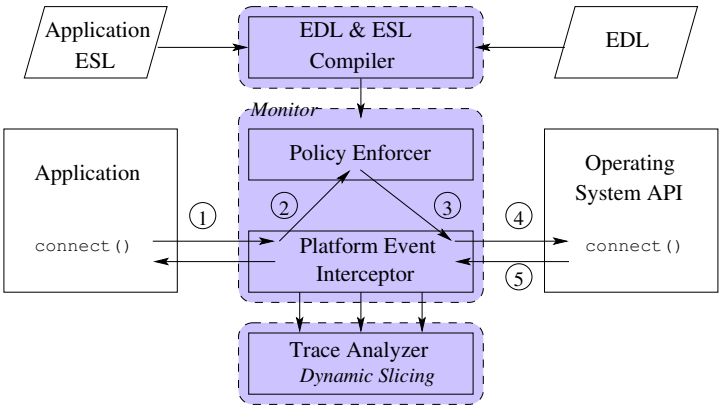


Figure 9: Architecture diagram

- There is an edge from (v, e_i) to (v', e_k) (denoted as $(v, e_i) \rightarrow (v', e_k)$) iff (v', e_j) depends on (v, e_i).

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).

3.3.1 Policy Enforcement

Policy enforcement occurs at run-time. Once a security policy is constructed for an application, that policy is compiled into a run-time DLL and is loaded into the enforcement monitor. We follow the standard sandbox approach to implement the monitor. Figure 9 shows the implementation architecture of our system. An enforcement cycle can be divided into five stages, shown in the Figure.

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 EDL event* by the *event-interceptor DLL* and is passed to the *policy-enforcer DLL*.
3. The *policy-enforcer DLL* processes the abstract EDL event and decides on an appropriate action based on the security policy.
4. If the abstract EDL 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.

A detailed description of the system can be found in the technical report [34].

3.4 Open Problems

Handling Multiple Traces. Our current implementation does have some limitations. First, the trace analyzer cannot handle multiple, different, execution traces from the same program. A program's runtime

traces may change due to many factors, such as environment variables and user inputs. Therefore, it would be useful to analyze multiple traces at once, and extract the difference and similarities through the analysis. For example, if a program connects to two different remote servers on two separate executions, then the corresponding traces may contain very similar call patterns, but with different arguments. Learning the difference and similarities among multiple traces would greatly improve the effectiveness of the tools.

Query Interface. The primary objective of the framework is to assist users to understand program behaviors, both benign and malicious. Therefore, providing a user interface that is not only simple to use, but also useful is the ultimate goal of my research. To this day, I have implemented a simple GUI interface for analyzing the traces. However, a more powerful interface is needed. I will address this issue through designing and implementing a *query interface*.

4 Addressing Type II Privacy Violations (Information Leakage)

A major drawback of access control-based approaches for combating privacy violations is that an authorized application can still leak protected information to unauthorized entities. In other words, discretionary access control cannot address Type II privacy violations (i.e. information leakage). For example, let us assume that a user confines a web browser inside a sandbox so that the browser can only access its cache and configuration files, and make network connections. Although the browser cannot read other files owned by the user, it still can access its cache files, and send statistical information regarding the cache content, such as web sites visited and the user's web surfing pattern, to a malicious remote web server. This information leakage cannot be detected by discretionary access control, such as described previously.

Information flow control complements discretionary access control by analyzing dataflow inside the software, and hence can detect privacy violations .

Prior work [29, 3, 21, 39, 48] in information flow control has focused on applying static analysis techniques on source code, but little work has been done in addressing information flow dynamically, on binaries. This is mainly due to the high cost of analyzing information flow at runtime, and special hardware support has been proposed to remedy the problems [24, 25]. For instance, Fenton's Data Mark machine uses special hardware to track the security level of register content [24], but it is not clear if such a system was ever built.

I will examine software-based techniques to track and enforce information flow in *COTS binaries, at run-time*. In this section, I first describe the challenges of run-time information flow control. Then I propose a solution based on *tagged memory* [50, 28, 41, 16], a technique used by program runtime verification community for addressing memory- and type- safety issues in both C programs (source code) and binaries. I expect that tagged memory can also be applied to address secure information flow in binaries. However, it is not a straight forward process to adapt this technique due to its high runtime overhead. I will discuss some of the issues I expect to address.

4.1 Challenges of Runtime Information Flow Control

Unlike static information flow control, where source code is available for analysis and performance is not a major concern, detecting and controlling information flow at runtime must deal with binaries, where type information is not available and performance is a crucial factor. Analyzing information flow at runtime means both *registers* and *memory locations* must be tracked, and this could lead to both computation and storage overhead. We use a simple example, shown in Figure 10, to illustrate these challenges.

Figure 10 demonstrates secure information flow inside a simple program, where (a) shows the source code, (b) and (c) show two possible (pseudo) assembly code corresponding to the source code. The two variables, h and l, are marked with *high* and *low* security levels, respectively. The actual content of the

```
l = 0;
if ( h != 0 ) then
   l = h;
```
(a) Source code

```
        xor R1, R1   ; l
        cmp R0, 0    ; h
        jne next
        mov R1, R0
next:
```
(b) Assembly code 1

```
        mov [0x7777], 0   ; l
        cmp R0, 0         ; h
        jne next
        mov [0x7777], R0
next:
```
(c) Assembly code 2

Figure 10: An example of secure information flow in binaries.

variables are not important in this example. Here, h can leak its information to l if the *then* branch condition is true, hence violating privacy.

Figure 10 (b) shows one possible assembly sequence, where both variables are stored in registers. In this case, we need to track the register contents in order to monitor secure information flow. Figure 10 (c) shows another scenario, where h is stored in a register and l in a memory location. Clearly, in this case we need to track data flow in both registers and memory locations.

The example shown in Figure 10, though very simple, reveals the potential *performance and storage overhead* we need to address when analyzing information flow at runtime. Since both registers and memory locations must be tracked at runtime, there is a considerable amount of computation and storage overhead as a result of the bookkeeping.

In summary, I envision the following two challenges in dealing with information flow control at runtime.

- **Performance:** Dynamic information flow control requires the tracking of both registers and memory locations at runtime. This introduces potentially high computation overhead which we must deal with.
- **Scalability:** COTS binaries can be very large, and may use arbitrary amount of memory at runtime. Keeping track of information flow at runtime therefore requires approaches that are scalable.

4.2 Tagged Memory-based Information Flow Control

Background on Tagged Memory. Tagged memory is a standard run-time technique used to address both memory- and type- safety problem in software. The basic idea is to tag additional bits of information, such as *validity* and *type information*, for each memory location. At runtime, the tag bits are used to track various information, such as allocation status and type information, about each location. Before an instruction is executed, the runtime system must first check the corresponding tag bits to ensure that the instruction will not violate any of the memory- or type- safety properties.

Yong describes memory-safety enforcer (MSE) [50], a tagged memory-based system that detects memory-safety related errors at runtime. In MSE, each byte of memory is tagged with one bit to indicate the validity of the memory location. A memory check is performed before every access to a memory location. If the corresponding bit is tagged invalid, then a memory access violation is said to have occurred. Purify [28] also uses tagged memory to detect memory-safety errors and memory leaks. At compile time, Purify instruments object code so two bits are tagged to each byte of memory: one for indicating whether the location is allocated or not, and the other for indicating initialization status of the location. At runtime, Purify use the two bits to track memory accesses and hence detect memory violations.

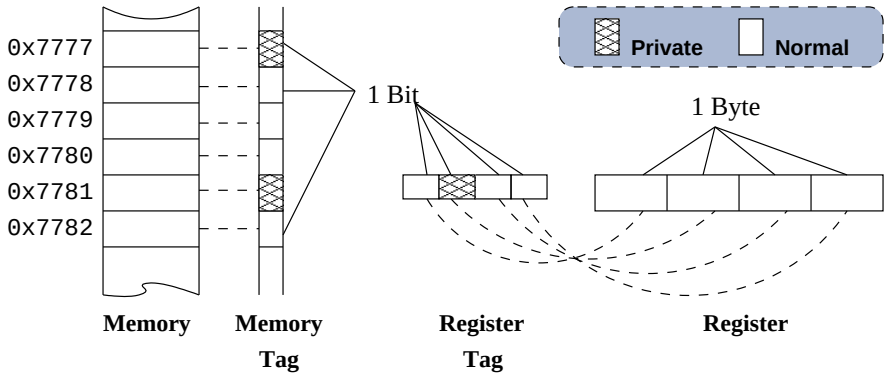


Figure 11: Using tagged memory to monitor secure information flow.

Both MSE and Purify addresses memory-safety issues in C programs and hence require compiler support. Valgrind [41] and Hobbes [16] deal with memory-safety in binaries. They both use run-time interpreters to emulate the CPU. Binary programs are intercepted at load time and each instruction is executed through the interpreter. Valgrind uses tagged memory to enforce memory-safety at runtime; Hobbes also checks for type-safety in addition to memory-safety. Since both Valgrind and Hobbes are based on interpreters, performance overhead is a major concern: both systems report over $40\times$ slowdown in execution time [3].

Tagged memory can also be used to address type-safety in software. Yong and Horwitz [51] and Loginov et al. [37] adapt tagged memory in their Runtime Type Checker (RTC). RTC uses four bits to encode type information for each byte of memory location. This information is obtained through source code. A runtime system detects type mismatches by checking the type of a memory location against the expected type for each instruction.

Applying Tagged Memory to Information Flow Control. We propose using tagged memory to track information flow in software. Instead of using one bit to indicate the validity of a memory location, we can use the bit to indicate the security level of the location. The bit is set to *high (*1*)* when the memory contains sensitive data, and *low (*0*)* otherwise. The entire memory is initialized to zeros initially; when a high-security data is loaded into a memory location, the tag bit is marked as high (1). At runtime, when an memory location is accessed by a function call, we first check the memory tag against the security level of the function. If the tag indicates high security content, and the function call is allowed to access high-security data (this is set by the security policy), then the call is allowed. Otherwise, the call is denied. This way, we guarantee that data from a protected resource cannot flow to unauthorized resource (such as a network socket that is connected to a remote site considered to be malicious). The left side of Figure 11 illustrates this technique, where each byte of memory has a one-bit tag associated with it.

Tagged memory-based approach can only address information flow in memory, but not in registers. In order to have a complete picture of information flow in software, we need to be able to track dataflow in registers as well. The Data Mark machine [24], for example, has a tag for each register in the system. However, their approach requires additional hardware support, and therefore is not applicable to our situation. Instead we need to investigate software-based techniques for tracking information flow in registers.

We can extend the memory tag approach to registers, by adding memory-based tags for registers as well.

[3] $40\times$ is Valgrind's worst performance; the best case is about $5\times$ slowdown.

For instance, assuming that each register is 32 bits, then we could use a four-bit tag for each register, where one bit is mapped to each byte of the register. This approach is shown on the right side of Figure 11. At runtime. all instruction streams are intercepted in order to correctly update the register tags according to the instruction. This is similar to a technique adopted by Redux [40], a dynamic dataflow analyzer built on Valgrind. However, Redux is designed to track actual runtime values, consequently it incurs extremely high overhead–in terms of both storage and computation. As a result, Redux can only track dataflow in small binaries, and it incurs the high runtime overhead from Valgrind–the framework it is based on. These drawbacks make this approach not suitable for our work since we have the requirement for both *performance* and *scalability*.

4.3 Open Problems

Runtime Overhead. As discussed above, existing techniques for tracking memory and register values at runtime incur very high performance overhead. However, this performance overhead is mainly due to the amount of bookkeeping these techniques must perform at each step. For instance, Valgrind provides a memory checker which uses *nine tag bits* for each byte: one bit for addressability and eight bits for validity. Memory checking means checking all nine bits and therefore is very time consuming.

Information flow control, on the other hand, does not have the same complexity requirement, and therefore can be solved in a simpler way. In secure information flow control, the key issue is whether a byte contains a high-security or low-security value. This information can be encoded using a single bit. Consequently, I expect to be able to find a solution with reasonable performance.

Parameterized Sensitivity. The objective of secure information flow control is to track the flow of secure, or private, information. An obvious, but naive, approach is to track information flow at all time, regardless of the type of the information (private or not). However, this is not necessary, since we only care when a private file is being open for read. Therefore, better results can be achieved by knowing *when to start tracking*. For example, a web browser might open a *tmp* file for write, or it might open a cache file for read. If the user only considers the cache file to be sensitive, then we do not need to track information flow when the *tmp* file is opened. In the second case, however, we will start tracking data flow from the moment the cache file is open. This issue, called *parameterized sensitivity*, can help to reduce the runtime overhead of information flow control. I will address this issue in my research.

5 Research Plan and Timeline

I divide my research plan into three areas: *discretionary access control*, *secure information flow control*, and *evaluation*. For discretionary access control, my main objective is to address some of the limitations of the framework we have developed. For secure information flow control, I will investigate existing techniques and develop new approaches. Finally, I will evaluate my implementations on real-world COTS software.

1. **Discretionary Access Control**

   We have implemented and tested the sandbox framework described in this proposal. However, the framework has some limitations which I plan to address.

   - **Handling multiple traces.** A single execution trace may not present a complete and accurate picture of software's runtime behavior since software's behavior may change depending on the environment and user inputs. Therefore, it is important to analyze a collection of execution traces of the same software for different run and uncover both similarities and difference among the

traces. This will help improving the refinement process, and hence make the sandbox framework more robust.

- **Query Interface.** Program traces may be very large, containing enormous amount of information. Sifting interested information from this large dataset requires a well designed *query interface*. Therefore, providing a user interface that is not only simple to use, but also powerful is the ultimate goal of my research. I plan to provide a query interface with an easy-to-use API. In addition, I also plan to refine the GUI frontend using the query interface.

2. **Secure Information Flow Control**

   I plan to address runtime secure information flow control in two phases: exploration and implementation. In exploration phase I plan to implement a prototype using existing techniques from both information flow control community and runtime program verification community. Then, I plan to apply the knowledge gained from building the prototype to provide a full implementation of runtime secure information control system for COTS software. The system will be tested on real-world COTS software.

   - **Exploration.**

     I plan to first investigate Valgrind, a runtime verification framework that has been used to address memory- and type- safety issues, among others. I plan to build a prototype of secure runtime information flow control around Valgrind. The objectives of this part are two. First, I want to validate the idea of using tagged memory to track secure information flow in COTS software, and Valgrind provides a suitable environment for achieving this objective. Second, through prototyping, I want to identify problems and obstacles in addressing secure information flow at runtime. More specifically, I will examine the performance bottleneck of the approach. This will help me understand some of the engineering challenges in addressing information flow control in binaries.

   - **Implementation.**

     Building on the knowledge gained from the prototype, I plan to investigate new techniques and enhancements for enforcing information flow control-based policies. Since Valgrind is designed as a generic framework for building program verification tools, it has to trade performance for functionality. My work, on the other hand, only focuses on enforcing secure information flow, therefore can be optimized for performance.

3. **Evaluation**

   The objective of my research is to develop practical techniques that can be used to *analyze* and *prevent* privacy violations in COTS software. To verify my work, I will evaluate the techniques on real world COTS software.

   Test Environment. I plan to use Windows systems as my primary evaluation platform because majority of the privacy violating software target Windows. This should provide me with a large selection of test cases. However, other systems, such as Linux, are also considered. I intend to make sure that the techniques I develop are platform-independent. Our dynamic slicing approach operates on abstract events (defined using EDL) that are platform-independent.

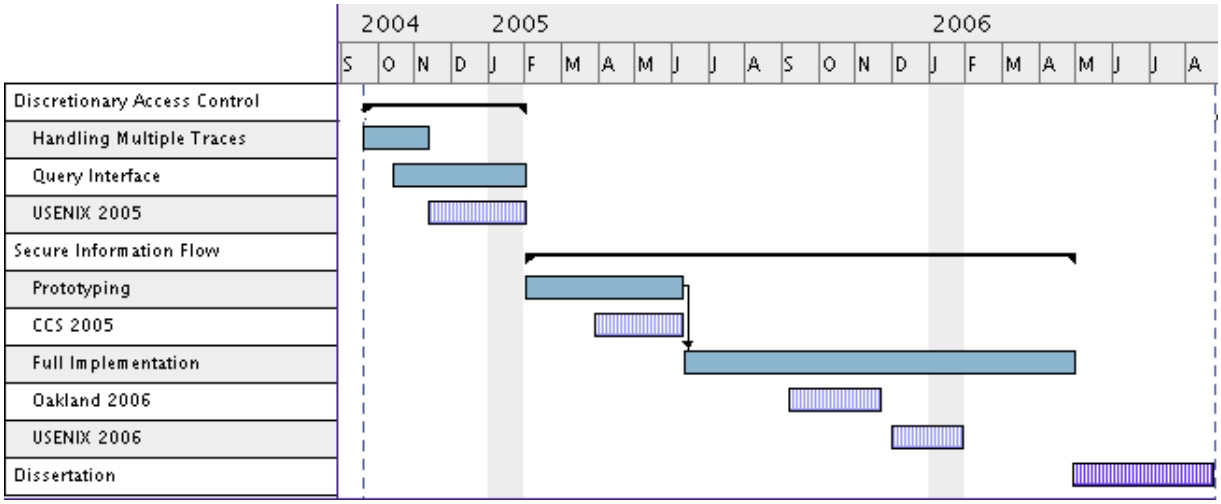


Figure 12: Research Plan and Schedule

Case Studies. My work focuses on addressing privacy violations in COTS software. I have initially selected two popular COTS software: Kazaa [1] and RealOne Player [2], as my test cases. The initial study performed on them reveals privacy violating features in both applications: Kazaa retrieves and displays advertisements from remote servers; RealOne Player sends information about the music being played back to its home server. I intend to continue using these two COTS software as my test cases.

In addition to Kazaa and RealOne Player, I plan to use Microsoft's Internet Explorer (IE) as another test target. IE is an interesting test case because it allows additional extensions (called Browser Helper Objects (BHOs)) to be installed inside IE. Furthermore, the BHOs execute in the same address space as IE, and hence have full access to the resources IE has. Many BHOs violate users's privacy by performing illegal activities while disguising themselves as *useful services* [18].

4. **Schedule**

The tentative schedule for my research plan is shown in Figure 12. I plan to finish my dissertation by the end of August, 2006.

References

[1] Kazaa. *http://www.kazaa.com* (circa August 2004).

[2] RealNetworks. *http://www.real.com* (circa August 2004).

[3] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, January 20–22 1999. ACM Press.

[4] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[5] Ad-aware. *http://www.lavasoft.de* (circa August 2004).

[6] H. Agrawal and J. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[7] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[8] Air Force Research Laboratory. Commercial-Off-The-Shelf (COTS): A Survey, Decembmer 2000. *http://www.dacs.dtic.mil/techs/cots/* (circa August 2004).

[9] G. Ammons. *Strauss: A Specification Miner*. PhD thesis, University of Wisconsin-Madison, 2003.

[10] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[11] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In E. Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, March 2004.

[12] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[13] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. March 2001.

[14] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 184–189. IEEE Computer Society, 1999.

[15] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the USENIX 1995 Technical Conference*, pages 165–175, New Orleans, LA, USA, 16–20 1995.

[16] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *Proc. European Joint Conferences on Theory and Practice of Software*, 2003.

[17] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[18] cexx.org. Counterexploitation. *http://www.cexx.org/* (circa August 2004).

[19] M. Christodorescu and S. Jha. Testing malware detectors. *SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[20] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[21] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[22] P. J. Denning. Third generation computer systems. *ACM Computing Survey*, 3(4):175–216, 1971.

[23] J. S. Fenton. *Information protection systems*. PhD thesis, University of Cambridge, 1973.

[24] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[25] I. Gat and H. J. Saal. Memoryless Execution: A Programmer's Viewpoint. *Software – Practice and Experience*, 6:613–615, 1976.

[26] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[27] G.S. Graham and P.J. Denning. Protection–principles and practice. In *Proceedings 1972 AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. AFIPS Press, 1972.

[28] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, 1992.

[29] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 365–377, New York, NY, USA, January 19–21 1998. ACM Press.

[30] Internet Security Systems. root shell. *http://www.iss.net/security_center/advice/Underground/Hacking/Methods/Technical/root_shell/default.htm* (circa August 2004).

[31] Jane's Defence Industry. The evolution of COTS in the defence industry, October 2000. *http://www.janes.com/business/news/jdi/jdi001009_1_n.shtml* (circa August 2004).

[32] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[33] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters (IPL)*, 29(3):155–163, 1988.

[34] L. Kruger, H. Wang, S. Jha, P. McDaniel, and W. Lee. Towards Discovering and Containing Privacy Violations in Software. Technical Report 1515, University of Wisconsin, Madison, WI, 2004. *http://www.cs.wisc.edu/∼hbwang/spyware/sandbox86-TR.pdf* (circa August 2004).

[35] B. W. Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[36] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[37] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 217–232. Springer-Verlag, 2001.

[38] G. McGraw and G. Morrisett. Attacking malicious code: Report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, September/October 2000.

[39] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symbosium on Operating System Principles*, pages 228–241, 1999.

[40] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[41] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[42] J. Palsberg and T. Zhao. Efficient and flexible matching of recursive types. *Information and Computation*, 171:1–24, 2001.

[43] N. Provos. Improving host security with system call policies. Technical Report 02-3, University of Michgan, November 2002.

[44] J. Saltzer and M. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.

[45] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[46] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Lecture Notes in Computer Science*, 2000:86–101, 2001.

[47] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages (JPL)*, 3(3):121–189, September 1995.

[48] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[49] D. A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.

[50] S. H. Yong. *Runtime Monitoring of C Programs For Security and Correctness*. PhD thesis, University of Wisconsin-Madison, 2004.

[51] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 307–316. ACM Press, 2003.

[52] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)*, 2003.