Dynamic Label Binding at Run-time

Yolanta Beres, Chris I Dalton

Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ, UK

yolanta. beres, chris.i.dalton
@hp.com

ABSTRACT

Information flow control allows enforcement of end-to-end confidentiality policies but has been difficult to put in practice. This paper introduces a pragmatic new approach for tracking information flow while the process is running at the same time applying dynamic label binding. The underlying implementation mechanism uses machine code instruction stream modification to track individual data movements and manipulations within the address space of an application. This gives the ability to precisely determine all information flow causing operations and apply controls that do not overly restrict what computations can be performed.

Keywords

Information flow control, labels, data labeling

1. INTRODUCTION

Protecting confidentiality of data manipulated in computer systems, especially in distributed systems, is an increasingly complex challenge. While business level security policies define protection requirements in terms of data, the actual enforcement of these policies is done through applications and through system and network configurations. Though they are important security building blocks, these standard techniques alone fail to adequately guarantee end-to-end data confidentiality. Standard techniques such as access control check that only authorized processes are able to access the confidential data. The controls are placed on the release of data but no checks are placed on its propagation. This fails to guarantee that confidential information will not be misused once it is released to authorized processes and users. Since no restrictions are placed on the propagation of the information once data is read from a file, for example, an authorized process may, through error or malice, improperly write data from that file to other, non-authorized, locations.

New Security Paradigms Workshop 2003 Ascona Switzerland

It is unrealistic to assume that all the applications in a computer system are trustworthy; applications are rarely bug-free, may contain logic errors in their design, and are increasingly hard to configure correctly. This means that either confidential information cannot be entrusted to applications or additional security mechanisms have to be introduced. To ensure that confidentiality of data is maintained protection mechanisms are necessary to track how information flows within the processes using it. This is expressed by information flow control. In this framework, it is assumed that computation using confidential information is possible, and that it is important to prevent the data from flowing to inappropriate destinations.

Information flow control is usually based on a notion of *labels* that allow information owners to express confidentiality requirements. In addition there are rules that must be followed to propagate labels as computation proceeds in order to avoid information leaks. Finally, flow policies dictate where and how information should flow in a system. Any approach for enforcing information flow control has therefore to address the following main questions: how labels should be bound and propagated on data and how data labeling together with flow policies should be integrated into a functional system.

Multilevel data labeling models that have been used in military environments to enforce mandatory access control, support mostly *static label binding* where the security label of the object is constant[1]. A system build only around this type of binding ends up unduly restricting how contents of the objects can be changed. In many practical systems the fixed labeling of many objects is a unrealistic requirement to comply with as it drastically restricts what type of computations can be performed in applications, resulting in only a limited number of applications allowed to run or requiring applications being highly customized.

In this paper we introduce the mechanisms that allow to track information flow within the application at runtime by analyzing the machine code of the running process. The mechanisms support *dynamic binding* where the security label of the object varies with its contents. With dynamic binding the labels are updated dynamically in the course of execution. We propose an implementation approach that uses machine code instruction stream modification to track individual data movements and manipulations within the address space of an application. This allows to precisely determining the origin of data as it flows within the executing process. We show that this approach does

[1] In some models [15] the label change may be permitted under certain, highly constrained conditions, and this mostly comes from the subjects requesting change of label on the objects that they have access to.

not unduly restrict the application behavior and being such, we believe, are thus more applicable to commercial environments.

The remainder of the paper is organized as follows. In Section 2 we describe previous related work in the area of information flow control. Dynamic label binding model is introduced in Section 3 that also describes the approach proposed for tracking information flows that occur indirectly through conditional structures. The challenges of implementing run-time data flow tracking mechanisms are discussed in Section 4. Finally, in Section 5 we describe the security architecture of our proposed implementation based on an application level instruction stream re-writing of machine code combined with enhanced operating system kernel functionality.

2. RELATED WORK

There has been much work on the information flow control, mostly on the static analysis of the program code, but in practical systems control over data propagation across domains hasn't been widely applied, except probably in military environments.

The majority of early implementations of information flow control type systems were based on the multilevel security model proposed by Bell and LaPadula [4], [5]. In this approach, each data item and each process is labeled with a corresponding *security level* that reflects a hierarchical confidentiality policy. The information flow control policy is enforced by a run-time mechanism that permits read access only if the security level of the process is higher or equal to the security level of the data item (no read up rule). Once the process has read data at one level, it cannot write data with a lower label (no write down rule). The mechanism works by augmenting the ordinary computation of data within a running program with a simultaneous computation of the corresponding label. This approach has been implemented in military environments and is prescribed by the U.S. Department of Defense "orange book" [12]. However it has proved to be too restrictive for general use since the results of computation usually end up to be labeled too sensitively for their intended use.

Fenton [14] proposed a run-time enforcement based on a finite state machine extended to include marks or labels on data items. The mechanism supported dynamic label binding but required enforcing memory-less execution property, where confidential inputs to a process should not remembered upon execution termination, which is not feasible in practice. In addition, the mechanism was only considered within an abstract computer model and has never been implemented although the idea of changing the process security label depending on the labels of the processed objects has been used within high [19] and low [16] watermark mechanisms.

Most of the later work concentrated on compile time mechanisms for information flow policy enforcement, first initiated by Denning's [10, 11] work. This approach tries to establish that an application is safe (i.e. would not violate the information flow policies) before deciding to run it by statically analyzing and amending the application *source* code. The information flow policies are specified and enforced using an enhanced type system at the programming language level (e.g. [3, 17, 23, 26, 29]). This allows creation of *security-typed languages* [25], where the types of program variables and expressions are augmented with annotations that specify either a security level or a security policy on the use of the typed data. The flow policies are then enforced by compile-time type checking. The major advantage of this approach is the ability to perform a rigorous analysis and to accurately track all possible information flows within an application by adding little or no run-time overhead. However in order to do so the method requires either access to the application source code (not always practically possible) or all applications be written in a secure-typed language (often too strong a requirement). Another potential weakness of using a compiler to validate information flows is that it places both the type checker and the code generator of the compiler into the *trusted computed base* (TCB) of the system. In addition, this approach does not work that well when labels need to be dynamically bound to data items. Source code analysis assumes that once statically checked the applications cannot be subverted, but it is not always a case as have been proved by Appel and Govindavajhala [1].

3. DYNAMIC LABEL BINDING MODEL

The aim of this work is to show that it should be possible to build a security system in practice that will track information flow within applications at run-time. We chose to work at the level of the machine code for a targeted application, both to minimize the size of the TCB and to affect as little as possible the application behavior. Most applications are not designed to manipulate the information labels and, therefore, we need to introduce the mechanisms that track information flows and calculate the labels as processes execute. In this section we present the dynamic label-binding model that specifies how labels should be changed as data is manipulated within applications.

In this approach we say that any object that can store arbitrary information has a label. The security labels are commonly used to describe how the information encoded into the object must be protected. They can specify the desired security level of an object or even encode data owner's policies on data handling [24]. For example, the label may indicate that information should never be sent over a public channel unencrypted. In this paper, however, we deliberately do not prescribe any meaning to labels, but briefly discuss the issue at the end of the paper.

With dynamic label binding the label of the object must be correctly updated each time information flows into it. Here the flows of concern are those that result from execution of sequence of operations that cause information to be directly transferred from one object to another, for example, assignment and copying. As the value of the object's label is not fixed, it must be changed in response to every data movement amongst the objects during the process execution. Whenever a new value is assigned to an object the object's current label is forgotten; instead it acquires the label of that value. When an operation is executed that causes information flow to an object from more than one other object, the labels of all the objects are combined and the object acquires the label that is result of that combination. This is in effect a restriction as the new label must be the same as or more restrictive than the old one. We can define this type of label change more precisely.

Given that $\{a,b,c,...\}$ is a set of logical data storage *objects*, which are containers of information and that $\{\overline{a},\overline{b},\overline{c},...\}$ is a set of *security labels*, the following ***re-labeling rule*** is valid:

If a value $f(a_1,...,a_n)$ flows to an object b then the security label of b is changed to $\overline{b}' := \overline{a}_1 \oplus ... \oplus \overline{a}_n \oplus \overline{b}$.

Here the operator "$\oplus$" is a label combining operator that is used in computing the security label of the result of a binary function on a pair of operands. For example, the label of the result of a binary function on objects a and b is $\overline{a} \oplus \overline{b}$. Extending this to an n-ary function $f(a_1,...,a_n)$ the label of the result is $\overline{a}_1 \oplus ... \oplus \overline{a}_n$.

Giving a confidential data to an untrusted process does not create an information leak as long as the labels of objects manipulated during process execution are computed according to the defined re-labeling rule. We say that information can be leaked only when it leaves the process through the write operation, so only at that point security policies should be enforced to prevent leaks.

3.1 Implicit Information Flows

The defined re-labeling rule is not sufficient to guarantee that labels are correctly updated under *all* flow causing operations. The operations of concern are conditional statements such as *if, while, for,* and *do while*. They transfer information from objects in the conditionals implicitly to other objects. This type of information flow is easily traceable at the well structured programming language level, as was demonstrated by previous work [3, 11]. However, at run-time only traces of executed machine code can be analyzed. This examination will not reveal the full program flow, and so it is nearly impossible to detect all data dependent on the conditionals while the process executes. As conditional structures can be fairly easily detected from program flow graphs in this section we propose to use the same technique for machine code by analyzing the flow graph of application's code at the time it is loaded.

We first describe implicit information flow by considering the following simple example:

```
b=0;c=0;

if (a==0){
        c=c+1;
}
if (c==0){
        b=b+1;
}
```

Assuming that value of a can be *0* or 1 at the end of this code b's value will always be equal to the value of a. Although there is no direct copying of a to b there is an indirect flow of the information through the two conditionals. Information from a to b would leak both through executing and not-executing operations `c=c+1` and `b=b+1`.

Given that labels of objects involved are correspondingly $\overline{a}, \overline{b}$ and $\overline{c}$, it is easily seen that these labels would stay the same by the end of this code because all operations are performed on a single operand. If a contains information labeled as "secret" and b is labeled as "public", then the "secret" value of a will be leaked through "public" b.

In this example the comparison expressions also carry away some information, but previous re-labeling rule does not take into account such flows. This type of flow occurred not as a result of direct transfer of value to an object, but as a result of executing or *not executing* an operation when that operation is conditioned on some value. We will call the flow of information that occurs through execution of operation as *direct implicit flow*, and the one that occurs through non-execution as *indirect implicit flow*.

To insure the correct label binding in the given example the label of c should have been updated to $\overline{c}' := \overline{c} \oplus \overline{a}$ independently whether c:=c+1 or a:=a-1 is executed. Similarly on the second conditional b's label must be updated to $\overline{b}' := \overline{b} \oplus \overline{c}'$, giving the final label of $\overline{b} \oplus \overline{c} \oplus \overline{a}$.

3.1.1 Static code analysis

In order to know what objects are affected by implicit flows we need to have information about all possible execution paths of the program, but it is clearly infeasible to create and evaluate that at run time. The most appropriate solution would be to analyze control flow transfers of the program prior to execution to determine what code blocks follow (or precede, as for the *do while* structure) the conditional statement. This information can then be used at runtime to update the labels once the memory and register locations of the affected objects are known. Binary code disassembly techniques [8, 9] can be used to construct a control flow graph (CFG) that represents abstract execution structure of machine code. Once the control flow graphs have been constructed the basic blocks can be analyzed for conditional jumps and loops.

The flow graphs for conditional structures such as *if* and *while* have a useful property of having a single beginning point at which the control starts and a single exit point at which the control leaves. For example, the code in figure 1 has two such structures, one nested within another. We exploit this property to identify the set of operations, values of which are implicitly affected by the conditionals on the entry point. All branches following a conditional have an implicit flow of information from the conditional. At the machine code level this is the value of a particular memory or register location. Therefore, when calculating the labels for branches following a conditional we need to take into account the label of the location in that conditional.

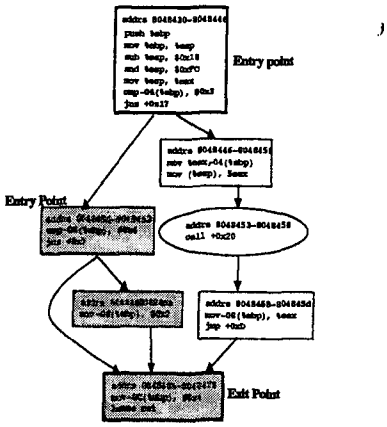

Figure 1. A control flow graph with two highlighted conditional structures.

In order to implement these schemes we need to add additional instrumentation code to the binary so that we can trace its actual control flow at runtime. We use the CFG to identify which code blocks are affected by the conditional and also to determine when we no longer have to take into account the label of a particular conditional branch, i.e. when that conditional can no longer cause an implicit flow.

During the static analysis the code is instrumented to provide additional information about the execution path taken[2]. This includes identifiers for entry and exit points of conditional structures, as well as of the blocks within the conditional branches. We say that the label of a particular conditional is no longer relevant when flow reaches the immediate forward dominator node of that conditional branch node in the CFG.

The construction of the control flow graph and static code instrumentation can be performed ahead of time or at least at load time thus reducing run-time performance overheads. It must be said that extracting accurate CFGs for certain binary code can prove challenging, particularly so for IA-32 binaries since code and data can be mixed together as well as the problem of indirect conditional jumps. Exceptions and signals also add to the problems of accurate CFG generation. Potentially if we can't determine the CFG for a particular binary we can choose to either not run it or assume worst case and apply the conservative approach of overly restrictive controls that we briefly describe at the end of the next section.

3.1.2 *Updated Re-labeling Rule*

The re-labeling rule defined previously at the start of the section 3 has to change so that labels in the conditionals are taken into account when the new labels are computed. At runtime, we can implement this by introducing the notion of the program counter (PC) of a process p and associate label $\overline{p}$ with that counter. This label reflects the current execution structure of the process and represents the labels of the entries to conditional structures. The value is determined by using a simple stack-based mechanism.

Whenever a conditional or loop entry point is detected the current label $\overline{p}$ is pushed further on the stack and the label of a conditional expression c is added, resulting in a new tag $\overline{p} \oplus \overline{c}$. If a statement is conditioned on the values of n expressions $c_1,...,c_n$ then the labels of these locations are first combined $\overline{c}_1 \oplus ... \oplus \overline{c}_n$ and the end result is combined with $\overline{p}$.

During all operations from the entry point the labels of the locations in branching expressions are updated by taking into account the current label of the program counter. The re-labeling rule introduced previously now takes the following form:

If a value $f(a_1,...,a_n)$ ***flows to an object b and the current label of the process counter is*** $\overline{p}$ ***then the security label of b is changed to*** $\overline{b}' := \overline{b} \oplus \overline{a}_1 \oplus ... \oplus \overline{a}_n \oplus \overline{p}$.

[2] In cases where we encounter back arrows in the CFG such as during *do while* loops and back-forwarding *goto* structures we also include support for back-tracking.

The labels are updated accordingly for all memory and register locations that are encountered after the conditional. When the node is reached that, according to the CFG, is the immediate forward dominator of the conditional branch node, the current PC label is popped off the stack and hence its value is restored to what it was before the conditional was encountered.

Consider the following simple example: `if (x==7) y = z;` during the assignment to y in the conditional branch, the memory location of y will get the label of z together with the program counter label that in this case includes (amongst possible others depending upon the structure and flow of the program) the tag of x. By doing this, the tag of y reflects both the direct assignment to it from z but also the implicit flow from the conditional on x.

Figure 2 shows the CFG corresponding to this simple example. The machine code basic block BB_i contains the instructions that cause the conditional branch based on the value of x. BB_j contains the machine instructions for the assignment to y and BB_k contains the code directly after the end of the *if*-conditional.

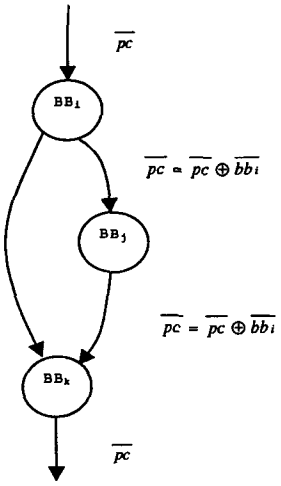

Figure 2. The program counter label.

The described program counter approach is similar to the theoretical model of data mark machine proposed by Fenton [10]. It takes into account the labels of locations in the comparisons and correctly propagates them across all operations that are executed after the comparison until the exit is reached. However, the approach does not capture indirect implicit flows that occur by not executing conditional branches. This is due to the fact that at run time the knowledge of the expressions that are not executed is lost.

Going back to the example introduced at the start of the section 3.1 we can see that based on this approach after entering the first conditional the program counter label is updated to $\overline{p} = \overline{a}$ (we assume it's empty when this code first starts). If $a = 0$, then the first branch is executed and the label of c is

updated correctly: $\overline{c}' = \overline{c} \oplus \overline{p}$. However, during the next branch, if c equals 0 then b's value becomes 1. If c does not equal 0 then no assignment happens. Thus, in this case the value of a is leaked through b, but its label still remains the same.

A potential solution is to revert to a more restrictive approach, as taken by the compile-time type-checking systems, as well as Aries project [6]. In this approach a `write` to a particular location within a branch is completely disallowed when the label associated with that location is equal or less restrictive than the label of the PC during that branch. This ensures that nothing is written within the branch that could not be written to outside of the branch – the fact that the branch is entered or not entered then gives no information away. In doing so, of course, one has to decide how to communicate the failure of the `write` operation so that no information is leaked through this communication as well.

We believe that taking the overly restrictive approach would preclude many applications from executing correctly, and might potentially still leak some information through other covert channels such as debugging errors, or failures to terminate. Therefore, in practice when running real applications it is acceptable to ignore certain information leaks. We discuss this and other covert channels in section 7.

4. INTERPRETING THE MODEL

As we have previously stated, our aim is to show that it should be possible to build a security system in practice that will support dynamic label binding and possible enforcement of information flow policies based on these labels. The resulting system has also to satisfy the pragmatic need to work on real applications without access to the source code of those applications or dependence on the code certification prior to execution, and to demonstrate an acceptable performance.

In attempting to build this system, we found that the primary difficulty with run time information flow analysis lies in detecting and monitoring all information flow causing operations in a running process.

A desirable proposition is to use the operating system where processes run as a *reference monitor* to apply object re-labeling rules and to potentially enforce policies that would be associated with the labels. A reference monitor can observe the execution of a targeted process and can halt it whenever it is about to violate a defined security policy. Most operating systems already fulfill the function of a reference monitor for such purposes as access control. For example, when a process attempts to access a file, an operating system intervenes to check that the process has necessary access rights before it is granted access.

The range of security properties that can be enforced using a reference monitor implemented in the operating system are limited, however, to the amount of events that are visible and traceable at this level while processes are running. Rules that govern system calls, for example, are feasible because the operating system can easily monitor them. But system calls are not used for all operations carried out by a process. When operations are executed within the processes own address space the operating system has no control over them.

Whilst it is true that all system resource access and all input and output operations that a process performs have to go through the operating system via the system call interface, any manipulation and copying of objects within a process's own memory space is largely invisible to the operating system. This leads to *blind spots* in the process behavior that cannot be monitored. It is possible to provide limited software watch points at the kernel level using a technique of *memory watch point triggers*. In this approach the features in the machine CPU hardware are exploited to set watch points over arbitrary memory locations and to trigger a trap on reads or writes to those areas that the OS can intercept. However, typically hardware only supports a few watch points of limited address range and these are shared across all processes[3].

An alternative to placing the monitoring mechanisms in the operating system is to merge them with the targeted application. The technique of *machine code re-writing* can be used to modify the original the machine code of an application / process either on the fly via dynamic instruction stream modification or statically by modifying the application object code before runtime. The SASI [27] prototype demonstrates the latter approach by embedding

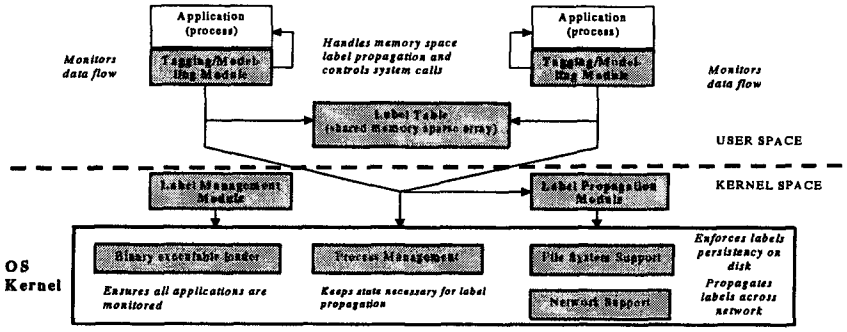

Figure 3. The security architecture of the proposed implementation system.

[3] IA-32 is fairly basic in this respect; IA-64 and UltraSparc offer much better support.

security automatons between each of the original machine code instructions of an application.

Dynamic labeling, however, can only be accurately interpreted and tracked at runtime, thus influencing us to choose the dynamic instruction stream modification approach. At the individual machine code instruction level we do have access to the points where a process reads and writes memory locations and registers and where we can insert additional instructions for propagating the labels. The dynamic instruction stream modification has been successfully used in the past for dynamic optimization [2, 20] and to provide virtualization support on non-naturally virtualizable platforms (VMware / Plex86 for IA-32 [28]).

5. PROPOSED IMPLEMENTATION FRAMEWORK

The basic security architecture of our implementation prototype where dynamic instruction stream modification is used to track information flows and to compute the labels is outlined in figure 2. In this framework it is possible to associate a security label with each byte of data in the various data sources on the system, such as files or network packets. When a process copies some data from a data source into its memory space, by doing a `read` system call for example, the label is bound to the copy of the data item that is now within the memory space of that process. As the process manipulates the data around its memory space the corresponding labels are also copied and combined based on the dynamic re-labeling rule defined previously.

5.1 Re-labeling with Machine Code Re-writing

The main component in this system is the tagging/modeling module responsible for making sure labels are injected into the memory space of a process and to also ensure that they are propagated around the address space of that process as that process uses the data.

At process load time we allocate, in shared memory, a sparse array that can potentially hold a label value[4] for each addressable byte in the memory regions assigned to that process. We also label registers so that a copy from one memory location to another location via the use of an intermediate register (as is often the case) also maintains the correct label.

The labels are updated only at the point where a process tries to export data from its address space, e.g. via a 'write' system call. Between the system calls we record what memory and register writes a process makes, and based on this we model the effect of those writes at a system call boundary - this allows us to lessen some of the performance impact.

Figure 4 shows a more detailed low-level architecture for machine code instrumentation and for computation of labels. As described previously, before run-time a process has to be statically instrumented so that it produces a trace of any memory and register write operations it carries out. This also informs what the program counter label should be as the write operations are carried out.

[4] Currently label values are 1 byte long.

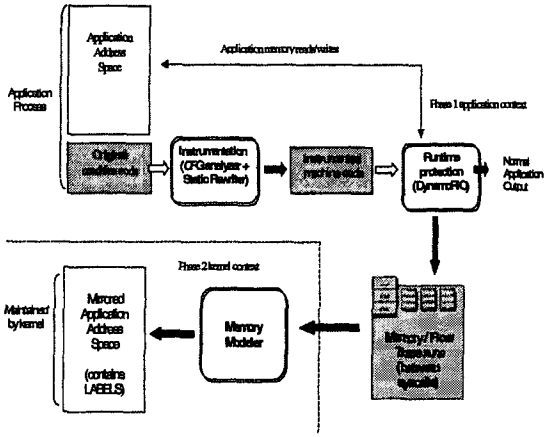

Figure 4. A low-level architecture

At runtime, the instrumented machine code is run under the dynamic instruction stream modification framework (DynamoRIO)[5]. This again involves re-writing the machine code but this time it is done dynamically, in order to ensure that the instrumentations are not bypassed.

When a process reads bytes from a data source (such as a file) into its address space via a system call, we add machine code to make it run an additional system call to determine the kernel maintained label values for those particular bytes in the data source. These label values are loaded into the sparse array for the locations within the process address space that the data was read into.

At a certain point, usually at the time of a system call, the tagging/modeling module is invoked to update the label values of the memory and register locations within the process. Given previously known labels for these locations and given a trace of machine code instructions (such as `mov B,A`) that cause a write from one area of the process address space (or register) to another area of the address space (or register) as well as instructions (such as `add` or `sub`) that cause data to be combined we are able to compute the new label values according to the previously described re-labeling rules.

When a process attempts to write data outside of its address space (via a system call) we re-write the operation so that the process first makes a system call to the kernel passing the label values of the data it is trying to write. At this point the kernel can be instrumented to check whether any particular policy, such as access control, applies on the passed label values. In cases when the policy prohibits writes to the intended destination the original system call is skipped over and an error call is returned to the process.

[5] Readers are directed to [13] for full description of the DynamoRIO system.

5.2 Kernel Support

In this framework, the mechanisms of application level instruction stream re-writing are combined with enhanced operating system kernel features to provide the necessary functionality.

One of the mechanisms that are provided at the kernel level assures initial data source labeling. The functionality used is similar to that already provided in implementations of multilevel security (MLS) systems that support file and network labeling [18]. In our case the mechanisms are able to provide support for individual labels per byte within data objects such as files. Our current prototype of data source labeling is based on the Linux operating system. For file system objects we have added label structures to the in-memory I-node kernel structures. For persistence and recovery there is a non-visible backing file stored on disk. We have not needed to change the on disk I-node structure – currently we support EFS2 and EFS3 file systems. The current prototype requires modifications to the Linux kernel source, but we believe that by making use of the Linux Security Module (LSM) [30] interface we can avoid the need for kernel source modifications in the future.

6. DISCUSSION

6.1 Security Policies

The system as presented in the previous sections has no built-in security policy model. Having a general-purpose information flow tracking mechanism is advantageous as it can be re-used to enforce different types of security policies.

The policies should dictate what information protection rules govern the labeled data. These rules are then applied on the system call as the process tries to transfer the labeled data out of its address space. Based on the label value and the applicable policy the operation is either permitted or denied. Having tracked the information flow whilst the application executes, the tagging module is able to communicate the accurate information label to the OS.

The label type and combining function, such as "$\oplus$" operator, should also be defined as part of the security policy model. For example, a user could apply a Bell and LaPadula style policy and then the label and function would define a lattice. This requires the operator to be defined as a comparison function that is reflexive, transitive, and anti-symmetric.

McHugh and Good [21] have taken a similar general-purpose approach by creating a simple information flow tool that can be used to verify information flow properties. However, their tool works only on the programs written in Gypsy programming language. In our approach we have gone one step further in attempting to create the tool for analysis of an arbitrary machine code without requiring changes at the programming language level.

6.2 Covert Channels

The dynamic label binding approach may not be sufficient to guarantee security of data in certain cases with very strong requirements, as it can be exploited for covert channels. We believe, that in order for a runtime information flow tracking mechanism to be used in a computer system to protect the information, the strength of guarantee must be sufficient to counter perceived threats. Therefore, the justification for ignoring certain potential information leaks must be a risk assessment, which has determined that the vulnerabilities present an acceptable risk. In many commercial organizations the acceptable risk value might be very different from what is acceptable in military environments, for example. As John McHugh correctly observed [22] "with a few exceptions, mostly dealing with small, very sensitive, information objects (such as long lived encryption keys), small information leakages are not of much concern today; after all, most systems are so vulnerable that it is far easier to take ownership of the system via a simple exploit than it is to attempt to signal information through the protection state". For many computations, some amount of information leakage is also both necessary and acceptable. For other computations, mostly operating on a highly sensitive data, information theory techniques can be used to estimate the amount of the information that would be leaked [7].

7. CONCLUSIONS

In this paper we have described a novel approach to run-time information flow monitoring within applications. This approach is based on *dynamic label binding* where the security label of the object is updated in the course of execution and varies dependent on its contents. We also propose an implementation approach that uses machine code instruction stream modification to track individual data movements and manipulations within the address space of an application.

In this approach the information is considered to be leaked only when it leaves the system where process runs, through write system call for example. The mere manipulation of data during the process execution is not considered to be release of information. The defined re-labeling rule ensures that as information flows the labels on data are updated to correctly reflect these flows.

8. REFERENCES

[1] A. Appel and S. Govindavajhala. "Using Memory Errors to Attack a Virtual Machine". In *Proc. of IEEE Symposium on Security and Privacy,* 2003.

[2] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A transparent runtime optimization system." In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[3] J.-P. Banˆatre, C. Bryce, and D. Le M´etayer. "Compile-time detection of information flow in sequential programs". In Proc. of European Symposium on Research in Computer Security, 1994.

[4] D. E. Bell and L. J. LaPadula, "Secure computer systems: A mathematical model." Tech. Rep. MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973.

[5] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations." Tech. Rep. MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[6] J. Brown, T.F. Knight, Jr. " A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information Flow". Project Aries Technical Memo ARIES-TM-015, MIT, November 2001.

[7] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Analysis of the Leakage of Confidential Data." *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 3, 2001.

[8] C Cifuentes, D Simon and A Fraboulet, "Assembly to High-Level Language Translation". In *Proc. of the International Conference on Software Maintenance*, November 1998.

[9] C Cifuentes and A Fraboulet, "Interprocedural Data Flow Recovery of High-level Language Code from Assembly". Technical Report 421, Department of Computer Science and Electrical Engineering, The University of Queensland, December 1997.

[10] D. E. Denning, "A Lattice Model of Secure Execution." In *Communications of the ACM*, vol. 19, no. 5, May 1976.

[11] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow." *Communications of the ACM*, vol. 20, no. 7, July 1977.

[12] Department of Defense, Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD (The Orange Book) edition, Dec. 1985.

[13] DynamoRIO System Overview. http://www.cag.lcs.mit.edu/dynamorio/doc/DynamoRIO.htm

[14] J. S. Fenton, "Memoryless subsystems." *Computing Journal*, vol. 17, no. 2, pp. 143–147, May 1974.

[15] S. N. Foley, Li Gong, and X. Qian, "A Security Model of Dynamic Labeling Providing a Tiered Approach to Verification", In *Proc. of IEEE Symposium on Security and Privacy*, 1996.

[16] T. Fraser, "LOMAC: Low Water-Mark Integrity Protection for COTS Environments." In *Proc. of IEEE Symposium on Security and Privacy*, May 2000.

[17] P. Herrmann, "Information Flow Analysis of Component-Structured Applications." In *Proc. of 17th Annual Computer Security Applications Conference (ACSAC'01)*, December 2001.

[18] Hewlett-Packard Co. (1996) HP-UX 10.16 CMW Security Features Guide.

[19] A.K. Jones and R. J. Lipton, "The enforcement of security policies for computation." In *Proc. of 5th Symposium on Operating Systems Principles*, November 1975.

[20] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution Via Program Shepherding." In *Proc. of 11th USENIX Security Symposium*, August 2002.

[21] J. McHugh, D. I. Good, "An Information Flow Tool for Gypsy". In *Proc. of IEEE Symposium on Security and Privacy*, 1985.

[22] J. McHugh, "An Information Flow Tool for Gypsy: An Extended Abstract Revisited". In *Proc. of Annual Computer Security Applications Conference*, December 2001.

[23] C. Myers, "JFlow: Practical mostly-static information flow control." In *Proc. of ACM Symposium on Principles of Programming Languages*, January 1999.

[24] C. Myers and B. Liskov, "A decentralized model for information flow control." In *Proc. of ACM Symposium on Operating System Principles*, October 1997.

[25] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security." *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, January 2003.

[26] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multithreaded programs." In *Proc. of IEEE Computer Security Foundations Workshop*, July 2000.

[27] F. B. Schneider and U. Erlingsson, "SASI Enforcement of Security Policies: a Retrospective." In *Proc. of New Security Paradigms Workshop*, 1999.

[28] J. Sugerman, G. Venkitachalam, and Beng-Hong Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor." In *Proc. of USENIX Annual Technical Conference*, June 2001.

[29] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis." Journal of Computer Security, vol. 4, no. 3, pp. 167–187, 1996.

[30] C. Wright and C. Cowan, *J. Morris, S. Smalley, and G. Kroah-Hartman,* "Linux Security Modules: General Security Support for the Linux Kernel." In *Proc. of 11th SENIX Security Symposium*, August 2002.