SECURING UNTRUSTED AND MALICIOUS SOFTWARE

INTRODUCTION

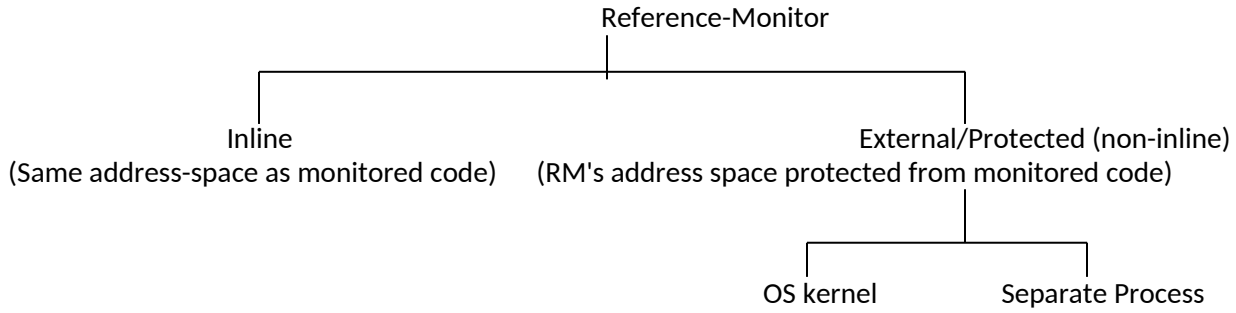
The first question that arises is “What do we mean by "security" and by "securing" software.” One way of defining the phrase "securing software" is enforcing a desired policy. This definition would then entail the capability of specifying the policy, which defines what this software can and cannot do.

Note that at the conceptual level, there is no difference between a policy for securing untrusted software from the ones studied earlier, say, MAC, DAC, Role-Based policies etc. However, when it comes to securing the untrusted software, there exists a need for finer granularity and deeper level of control than the standard operating system primitives.

Security policies can be enforced using the concept of reference monitor. The reference monitor should be protected from being compromised by untrusted code.

One of the properties that’s required of the reference monitor is that of "complete mediation". The requirement basically says that the reference monitor should be able to intercept/examine every operation that is being made by the code under inspection. The idea is that if certain operations are missed out, then it would possibly result in a failure to enforce the security policy.

There are two basic types of reference monitors (RM)



* Inline Reference Monitor (IRM) - IRM runs in the same address-space as the code on which the policy is to be enforced. The key issue here is that a mechanism is needed to protect the data and control flow used by the monitor from being compromised by monitored code.

* External/Protected Reference Monitor – In non-inline reference monitors, the reference monitor is in a protected address-space. That can be either within the OS Kernel or a separate address-space.

  The main distinction is that in an external/Protected RM, the monitored code cannot affect the data and/or the control flow of the monitor. While in the IRM, the monitor and the monitored code are in the same address-space and an explicit mechanism is needed for protection of the integrity of RM.

  However, do note there are advantages of IRM over non-inline Monitors:

  * Performance. In case of non-inline monitors, a context-switch occurs each time a check needs to be performed. This leads to a loss of performance in External monitors

* In non-inline reference monitors, the monitored code needs to be stopped when its data is in consistent state so that the necessary checks can be performed.

  This is not an issue when the checks happen once in a while. However with finer granularity checks, (say) checks after execution of every single instruction of the monitored code, the context-switches can be quite expensive and performance suffers. (On a typical processor, a context-switch involves few-thousand instructions.)

* The main question then arises as to at what granularity should the checks be performed.

For example, In StackGuard, every function call and return is monitored. Note that every function has only few tens to hundreds of instructions. Hence if we have context-switches at similar granularity, that is for function call and return, then performance would not be good (again because context-switches are far more expensive)

GRANULARITY W.R.T. EXTERNAL MONITORS:

MONITORING SYSTEM-CALL:

* Another feasible granularity level can be enforcing policy at system call level.

* Monitoring at system-call is a "natural" way of enforcing policy using a non-inline monitor. Following are some of the reasons:

1. RIGHT GRANULARITY

* When a system call is performed, there is already a context-switch in progress. The control is being transferred to the kernel. Checks at this junction would thus prevent additional overheads.

2. System calls are designed in a way that allows policies to be expressed easily.

(a) In a well designed OS, the number of system-calls is relatively less, measured in few hundreds.

For example: Linux has 319 system calls, FreeBSD has 330. Thus the number of operations that need to be monitored is reasonably small.

(b) System calls are designed to be orthogonal. There is just one system call for a specific purpose. Suppose, to the contrary, there are 10 different ways of doing the same thing. In such a case, if that thing is to be prevented, then the policy would have to examine all the 10 different methods

(c) Since the system-calls incorporate the concept of user-space and kernel-address-space, the issue of what data is relevant and what needs to be monitored is reasonably well-defined.

For instance, if a file is to be opened, the relevant data are the parameters to the system call. Hence there is no need to examine any other data in the monitored process's address-space. However, if process's data is monitored independently, we could have a TOCTTOU (Time of Check to Time of Use) issues. Especially in multi-threaded processes, where the check could ensure the correctness of data at a specific time, only to have the data changed by another thread by the time of use of that data.

(d) System calls are understood by a larger set of people than those that understand kernel internals. This means that more people are likely to be able to write system call policies, as opposed to policies based on deeper events within the OS kernel. For instance, the Linux Security Module (LSM) on Linux defines several security related events within the OS. This need for expertise makes it more difficult to write policies. (It should be noted, though, that if one has the expertise, then LSM can avoid several pitfalls associated with system call policies.)

MORE ABOUT THE RIGHT ABSTRACTIONS (Side discussion)

To clarify this point, consider the following example:

Example:

In SELinux, these monitoring operations have been moved deeper inside the operating system.

The basic goal of SELinux is the same. That is, enforcement of the security policy on user-level processes. In a way, this is a better approach, because it’s closer to the resources.

Moreover, on the system-call level, when a policy needs to be specified, there arise certain issues:

For example: name to object translation in the operating system. We might want to specify that a certain file should not be modifiable. At times we need to protect the name of the object, at other times we need to protect the object itself without any regard for the name.

This distinction is hard to express at the system-call level. Certain system-calls use file-name while others use file-descriptors. However when within the kernel, we can be more specific about the operation intended.

However the key point is that there are also disadvantages to such an approach.

DISADVANTAGES OF SELINUX APPROACH:

* There are a lot more operations inside the kernel and thus specifying a policy becomes more complicated.

* The policy (specified using SELinux) is very closely coupled to the operating system.

  SELinux uses a Linux Security Module which identifies the places within the kernel where the security checks should be performed and provides a framework/programming-environment to develop a reference monitor. Thus each time a security sensitive operation is to be performed; the reference monitor is called with all the relevant parameters.

  Thus when writing the reference monitor, we would need to know the points in the kernel where the security policy has to be enforced and understand what each interception point means.

As compared to this, then system call level granularity has the following advantages:

ADVANTAGE OF SYS-CALL GRANULARITY:

In comparison to the above, while writing policies at the system call layer, although some level of system knowledge is required (for example, file-manipulation in windows is different from unix), the system call interface generally tends to be the same across Unix-flavors of operating-systems.

Thus sys-call based policies are easier to write as compared utilizing hooks with operating system to enforce policy.

3. CAN BE DONE WITH EXISTING OS'ES

Another important point is that this technique can be implemented on existing operating systems.

System calls are basically a software interrupts that get dispatched to appropriate kernel routines. Thus, in principle, there exists a dispatch table to guide these system-calls. By replacing this table with another table of choice, we could effectively intercept all systems calls.

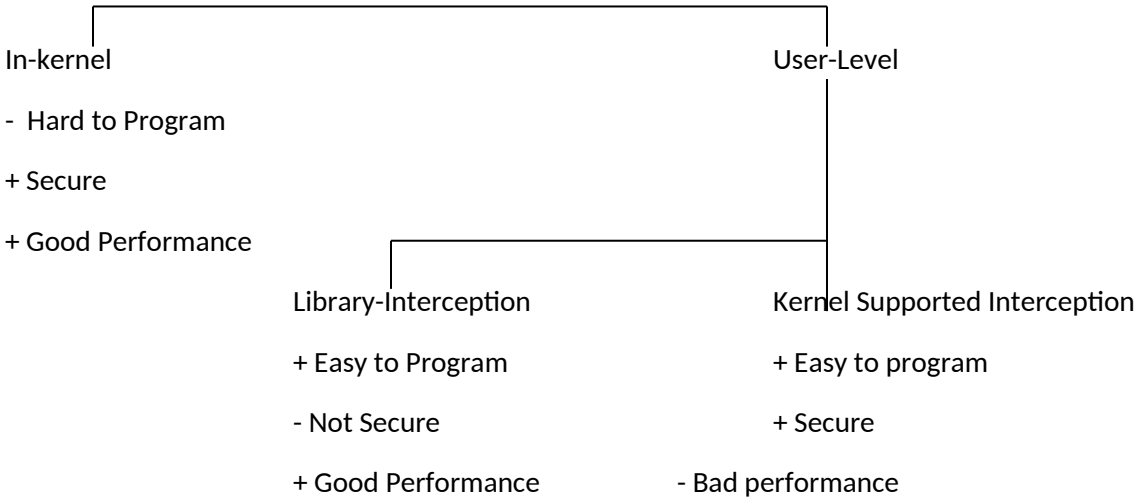
Thus there exists a backward compatibility in implemented a system-call level reference monitor

* For all these reasons, system-call interception, that is having system-call level granularity, is quite popular and a lot of research has focused on the use of this technique. On Linux, LSM is very popular as well; but on Windows, security tools rely much more commonly on system call interception.

MECHANISMS FOR SYSTEM CALL INTERCEPTION:

* The following is a classification for mechanisms for implementing system-call interception.

MECHANISMS FOR SYSTEM-CALL INTERCEPTION



I KERNEL-LEVEL SYSTEM-CALL INTERCEPTION:

* This method involved doing call interception in the kernel, say, by replacing the system-call dispatch table with one that routes the system call via the monitoring code for every system call. In fact, a number of security tools including anti-virus and application firewall products operate by doing system call interception at some level.

ADVANTAGE:

* Good performance. Since there is no context switch and just the data is being routed through another method in the kernel.

DISADVANTAGES:

* Kernel coding is much more difficult as opposed to user-level code.

* Operating Systems may not support replacing system call tables because this capability can be exploited by malicious code.

For example: Rootkits. The objective of a rootkit is to hide its presence from typical system monitoring tools, and to carry out their malicious activities while remaining hidden. Rootkits may be used to hide the presence of spyware or bot software on a system.

A popular way of implementing rootkits is to intercept system calls made by any process. So for example if any directory is being listed, the rootkit would intercept the data returned by the system call (used for directory listing) and remove any root-kit related files from the result. A similar strategy can be used to hide the presence of rootkit processes from process monitoring utilities.

Thus on Linux, the symbol that denotes this system-call table is not visible. Thus within the kernel, we cannot refer to this table by name. (But we can search for the table based on some kind of pattern matching on its contents.) On Vista too, the dispatch table was supposed to be protected, but this was relaxed later on since AV vendors complained.

II USER-LEVEL SYSTEM CALL INTERCEPTION:

1. Library-Interception:

* This method implies intercepting system calls by intercepting calls to the user-level library that makes the system call. This is because in a typical operating system, programs do not directly make system calls. Making a system call usually involves setting up the appropriate registers and executing the

software interrupt instruction. Hence it is easier to place this code in a library and for the user programs to call these library functions.

In Linux, this library is "libc". In Windows, similar role is played by "ntdll.dll".

* The basic idea is that while searching libraries, the OS searches for a specified set of directories. In Linux, the environment variable "LD_LIBRARY_PATH" stores the path of the directories to be searched for libraries. Thus by placing the instrumented library (which will intercept system calls and invoke the reference monitor on each call) earlier in the path, calls to libc can be intercepted.

  NOTE:

  Just as an aside, on Windows, the system call interface is not public and it not documented by Microsoft. Parts of the interface have been reverse-engineered and documented by third parties, but not the complete interface. Even for those operations that are documented, not all parameters are documented. The documented library is the Win32 library (kernel32.dll). As a result, there are many tools on Windows that rely on library interception of calls to this API.

ADVANTAGES/DISADVANTAGES of library interception:

* Performance. Since there is no context-switch involved, performance is adequate.

* Since this is at user-space, coding is relatively easier as compared to Kernel-level coding.

DISADVANTAGES:

* The main disadvantage is that this is not a secure mechanism. That is an untrusted code cannot be relied on to use this library. It could directly make system calls on its own. (On UNIX, such direct system calls are the most common means used by exploit code. On Windows, the use of Windows API is more common.)

* Even if the instrumented library is used, the checker is still running in the untrusted code's address-space and thus the code could again easily compromise the checks.

* As a result, this mechanism is inappropriate for use with untrusted code, or trusted code that contains an arbitrary code injection vulnerability. But it can be used on benign applications (which are trusted not to be malicious) that do not suffer from code injection vulnerabilities, or have been protected against injected code attacks.

  Consider a program written in Java or PHP, there do not exist binary code-injection vulnerabilities. However these might have injection of script-code. These are fine because launching of such scripts still requires benign code to make a system call or a function call. These operations can still be monitored --- since the benign code has not been compromised in any way before this call, it is reasonable to assume that it will not attempt to defeat library interception mechanisms.

  As a concrete example, consider an information flow policy that dictates that untrusted data should not be read or executed by a benign program. This means that there is no mechanism to exploit any vulnerabilities in this program to circumvent a reference monitor that enforces this policy using library interception.

EXAMPLE (OF LIBRARY-INTERCEPTION TECHNIQUE):

* In Windows, at times security tools including anti-virus tools use library interception method. Since the system-call interface is undocumented it becomes difficult to decide what should be allowed and what should not be allowed. Hence there is fair amount of usage of library-call interception method as a protection even against malicious/untrusted code, simply because having some protection is better than having no protection at all.

  Of course, the flip-side is that having a false sense of security brought about by such a protection mechanism may cause more harm than good.

2. Kernel supported Interception user-level:

* Many UNIX-based operating systems provide kernel support for performing system call interception at the user-level.

* For example: In Solaris, there is a “proc” mechanism, which helps with system call interception. In Linux, we have the "ptrace" mechnism.

  Ptrace is available in all Unix-based operating systems and had been originally provided for the purposes of debugging. This facility in Linux has been expanded to provide a way to intercept system calls as well.

* The basic idea is to have a "debugger" process examine the memory/state of the "debugged" process. Typical operations are "peek" (examining memory contents) and "poke" (modifying the state of the debugged process). Examining the registers of the process and single-stepping are other capabilities.

- The debugger can set "break-point" at desired points in the process under observation. This breakpoint can be set by writing (say) an invalid instruction at the desired breakpoint location. When the debugged process runs, when the breakpoint instruction is executed, a UNIX signal will be generated. Ptrace allows the debugger process to intercept this signal. At this point, it can examine memory interactively as desired by the user.

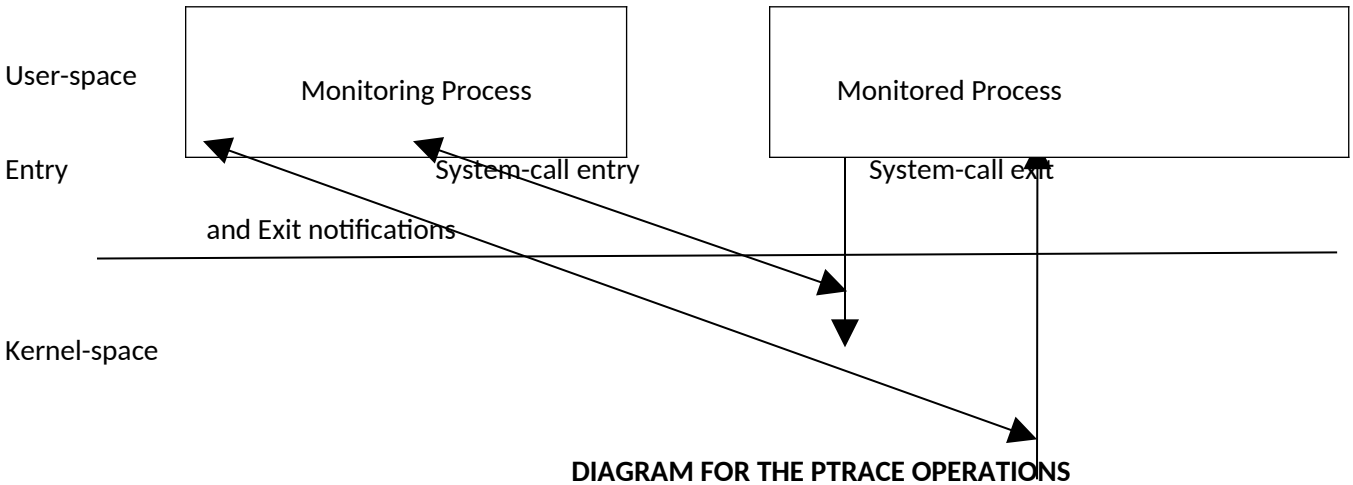


DIAGRAM FOR THE PTRACE OPERATIONS

* The ptrace interface has a couple of instructions which instruct/allow the monitored process to run until it makes a system call. When the process makes a system call, the control is handed over to the debugger/monitoring process.

  The monitor can then examine the system call being invoked, the parameters being passed to the system call and can even modify the register values and thus the parameters if it so wishes.

* Once the system-call is executed, the monitor is again notified of the result. Thus the monitor can again examine the results and also modify them if need be.

DISADVANTAGES of kernel-assisted user-level interception:

* The disadvantage of this method is the extra context-switches. For every system call there are two additional context switches (as compared to a purely in-kernel approach). Thus for processes that make a lot of system calls, the slowdown factor can be significant, sometimes more than 100%.

ADVANTAGES:

* The monitoring code is running at user-level and thus is relatively easy to implement.
* The mechanism is secure, and can be used with untrusted code.

III HYBRID TECHNIQUES:

* The basic idea of hybrid techniques is to improve performance by cutting down on the number of context-switches. Thus if the number of context-switches (in the ptrace mechanism) could be reduced by (say) a factor of 10, then performance would not be an issue.

* Thus if we can ensure that the most frequent operations are checked in the kernel and the rest of the operations are monitored at the user-level, we can achieve best of both worlds.

* Thus for example, we could have a policy that dictates that reads and writes are not security sensitive, and that only file open operations need to be checked. Thus reads-writes need not be sent to the user-level at all. On the other hand file-open operations can be sent to a user-level monitor.

  If reads-writes comprise 70-80% of the system-calls and only 20-30 % operations are file opens, then the improvement in performance is considerable.

INLINE REFERENCE MONITORS

ADVANTAGES

* Finer granularity
* Good Performance
* Much Better visibility:

  The basic concept is that if we are running in the same address space then its much easier to understand and interpret memory contents, as opposed to running in a different address space.

DISADVANTAGES:

* Separate mechanism to protect reference monitor's data/logic. Note that the traditional solution to this problem, protecting data and code of process, called for each process to have an address-space of its own.

* Many years ago a technique called SOFTWARE-BASED FAULT-ISOLATION was developed.

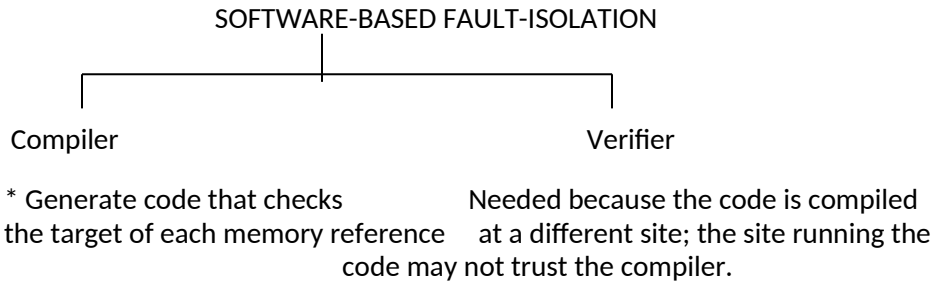
  In operating systems, in addition to security, there is another reason for having separate address-spaces. And that is fault-isolation. For example if one process corrupts its memory state, with separate address-spaces for kernel and other processes, those remain uncorrupted.

  For example: Windows-95 did not have separate address spaces for kernel and processes. This contributed to system instability.

  Hence the term "fault-isolation" is used synonymously with memory-protection

* The basic idea of this technique is that software-based techniques are used to enforce memory protection.

* The technique involved two components as follows:



The compiler would generate code such that before the monitored process performs any memory access, (say) a write, checks would be performed to ensure that the location of the write was permissible, (say) was not within a certain range (that is used to store IRM's data and/or code).

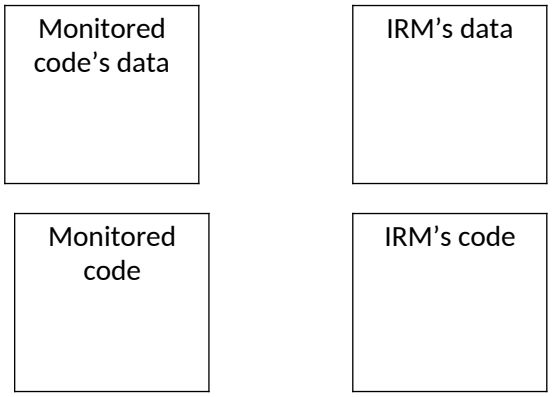
Another important point to note is that code could still evade the all the memory checks (say) by jumping over the memory checks and thus skipping the checks. Hence another important point is for the verifier to ensure that the code should not evade the checks by jumping over them.

The above requirement is a pretty difficult one. The software-based fault isolation technique was originally developed for RISC instruction sets where problems are generally simpler including the disassembly of binary code. The way the above technique was implemented in the original work was that a register was dedicated for all memory accesses. This register was always guaranteed to contain a valid address before any control-flow transfer instruction. Hence, even if jumps attempted to jump to a location after the check, it is OK since the register contents are guaranteed at the point of jump, i.e., it points to memory allowed to be accessed by the code.

An alternative approach is to statically analyze all control flow transfers and ensure that there are no jumps that bypass the memory checks. But this is hard to do --- there are indirect jumps, calls and returns, and their targets cannot be statically checked easily. (In the next lecture, we will talk about a technique called “control-flow integrity” that attempts to do this.)

Inline Reference Monitoring Techniques

In the last lecture, we started talking about **Inline Reference Monitors**. The idea is that the policy enforcement code runs with the same address space as the code being monitored. We can roughly divide the memory region into four parts as below.



Software Fault Isolation:

One of the most important things is that we need to protect the IRM's data from being modified by the monitored code because if it can modify the IRM's data, it can evade the policy and do whatever it wants. So what we said was that there is a way to instrument the code in such a way that the memory accesses made by the monitored code can be checked at run time in such a way that we can statically prove that monitored code can only write (or may be read) only its own data. We talked about a particular technique called **Software Fault Isolation** and the way it enforces is that it ensures that IRM's data will not be accessed by the monitored code. And basically what it does is:

- Check upper bound
- Check lower bound
- Perform memory access.

So essentially, every memory access performed by the monitored code is prefixed with two instructions that are designed to check if the reference is going into the IRM memory. If not going into the area, it is allowed to do the access.

Then there is a way to circumvent this, because malicious code can jump to the instruction to perform the memory access bypassing the checks. We need a mechanism so that this bypassing cannot be done. This could be done by reserving a register such that all data accesses will use that register and then by making sure that the register will always contain a legal value before any control transfer instruction or of course before every memory access. So any time a control transfer instruction is executed, if you can

already make sure that at the point jump is performed, the register already has a valid value or a value that will check the bounds, then it is fine even if the code arbitrarily jumps there because at the point where jump is performed you already know that the conditions are being satisfied. The basic idea of SFI is to introduce the notion of a dedicated registers so that these types of checks can be put into the middle of the code you don't trust.

In addition to protecting IRM data, we need to make sure the IRM code is protected and the entry into IRM code is regulated. Presumably the code in IRM has some sort of privileged level, and so the monitored code should not be allowed to invoke arbitrary code within the IRM. So we need to protect the IRM code keeping in mind that at some point the monitored code does need to invoke the IRM code. So there has to be some way to call and the call needs to be regulated. So how would we do this?

Gates:

We can use the concept of gates that we discussed in the context of switching between privileged and unprivileged execution modes. Gates enable the callers to tell that they want to enter, but they cannot say where they want to enter, for instance. So there can be one function in the IRM code, and that is the only entry point for the monitored code and everything else that need to be passed into, should be passed as data. (Side note: You need to think about the **Time of Check** and **Time of Use** sort of issues. If the parameters are passed through registers, that will remove time of check and time of use sort of issues.) We can generalize this case to permit multiple entry points, but still, it is the IRM that needs to decide the set of entry points that can be invoked by the monitored code.

Again the technique is used to insert these checks before a jump. The jump is either within the monitored code or if it goes outside the monitored code, it goes one of the allowed entry point of the IRM code. Again there are the same issue of jumping pass the checks and here also same technique will work. But since IRM code addresses are likely to be different from data addresses, you will likely need a second dedicated register to enforce again you need different registers to tell you what is valid for data access.

In this technique, the instrumented code will be generated by the compiler. But since this is generated at the site you don't necessarily trust, on the site the code will be executed together with the site supplied IRM, you have to actually check the instrumentation. They developed ways to do that and it is not so much complicated in case of RISC instruction sets where decoding instructions and disassembling is fairly simple because you have fixed length instructions and you have instructions thar are reasonably large (say, 32 bits) and jump targets have to be aligned on 4 byte boundary. These factors make it simpler and verification can be done easily. On the other hand, in case of complex instruction sets like instructions in x86 machines, the problem becomes a little bit harder. Plus, you don't have so many registers for dedicating registers.

Control Flow Integrity:

The notion of CFI is motivated very much by the example of jumping into arbitrary code to defeat some policy. But if you design properly, you can avoid it. But the question is: **Can anyone define some notion of CFI which talks about some integrity property regarding control flow transfers**

such that using this property you can build higher level security mechanism? For instance, we need two types of properties:

- One is to protect the IRM data, and
- The other property is to control flow from monitored code to IRM code.

So we need some kind of Control Flow Integrity such that IRM code will not be exploited by the monitored code. So some properties of control flow transfers should be enforced. We can enforce it by checking the property before making a transfer. CFI simply says to verify certain criteria regarding jump/call targets. Let us explain this with example. One of the policies for enforcement can be:

- **Target of control-flow transfer should be within the code segment.** It protects against code injection attacks. It is important to note that while doing this we are not making many assumptions. In contrast, in the context of buffer overflow defenses, we made several assumptions, e.g., the protected program will not actively try to subvert the defense. But here we have not made such assumptions here, still it guarantees that no code injection will be possible. (Note: there are some assumptions underneath, if you looked very closely at low level details relating to machine code. But the level at which we see, we don't need to have any assumptions to be made as long as we perform this check any time a control flow takes place.)

Does it protect from **return to libc** attack? The answer is no. Because libc is within the code segment and the check is rather coarse saying that the jump cannot go outside the existing code.

It is certainly a useful primitive because when you say you cannot execute injected code --- it gives you certain guarantees that you can build on. Let us talk about an example. Let's think about address space randomization. It is reasonably secure against the remote attacks. But let's say there is some vulnerability and there is some target that is not protected that allows someone to inject code. Notice that this not enough because of the randomization. Let's say the attacker's goal is to execute some system API function, say CreateProcess on Windows. Now he has to figure out where is that system function. He will not be able to directly call the intended function and execute his malicious activity as he does not know the location of this function (due to randomization). But the moment he has the injected code running, he can start scanning the memory and he will be able to find out the location of desired functions. There are lots of places he can look for, such as, there are linkage tables where these entry points for dynamically linked libraries are put, or he can simply scan the code memory, say, based on the contents of the first 16 bytes of a routine that may uniquely identify the routine.

Thus the idea is that, though there is address space randomization, but if there is one weakness in the system that allows injected code to run, then everything is lost. Injected code can now figure out all the randomness that is used. So the point is that if you couple the address space randomization with the technique like this (enforcing checks prior to jump and limiting within code segment), the fragility of address space randomization can be mitigated.

Thus you end up putting checks like this prior to each control flow transfer. However, as discussed before, malicious code may attempt to jump past the checks. Since we are only checking that the target is within code segment, it is possible for malicious code to jump past checks. However, in principle, the control-flow integrity technique should be able to protect itself: the threat being addressed is one of jumping to a location, and hence an appropriate CFI property should be able to prevent it. Let us consider the following criteria in this regard:

- a) **All calls should go to beginning of functions,**
  b) **All returns are to call sites known at instrumentation time**,
  c) **All jumps are to instructions in original code, except for jumps to control-flow transfer instructions --- these jumps to should instead transfer control to the checking instructions that precede the control-flow transfer.**

Here point (a) is trivial to understand. Point (b) says that any return should go to an instruction that follow a call. You might think it is better to enforce a stronger condition that returns go back to the call site; but enforcing this requires the CFI mechanism to maintain its own stack, which increases complexity. So we relax the condition here. We are simply saying that returns can go back to *any* call site. Point (c) addresses the question of jumping past checks, or otherwise attempting to interfere with the CFI checks. Note that 2(a) (b) (c) imply rule 1, provided all the targets are based on information that is statically available. (For instance, the implication won't hold if (b) were implemented by checking at runtime if the return address was preceded by a call instruction --- in that case, malicious code could create some code on the fly in data segment, and transfer control to that code by simply putting a call instruction before the desired target location.)

Implementation of CFI:

In terms of implementation issue, there are **direct jumps** and **direct calls**. Again there are **indirect jumps** and **indirect calls**. Direct means the target is specified in the code itself. For direct jumps, the check itself can be performed at the instrumentation time. Check need not be done at runtime and this is one reason why these techniques are reasonably fast. (Most jumps/calls are direct, so they dont need runtime checks.)

All indirect control-flow transfers, including returns, need to be checked at runtime, because the target of control-flow transfer are not statically known.

There are various ways to implement CFI.

Alternative 1: One simple approach is the one that **uses some sort of global Boolean array** denoting the valid addresses initialized at the beginning of the program. Thus every jump is checked against the target, and if it is fine, then the jump is made. The global array itself needs to be protected. You can do that by making the array read only. (If it is not read only, that is, if it is writable, then malicious code can modify the global array and can jump anywhere it wants.)

Limitation: This simple alternative has two limitations. First, it needs significant amount of memory, say, $1/32^{th}$ of address space (if all jump targets are required to be 4-byte aligned). Second, it assumes that the validity of the target is independent of the source address.

Alternative 2: The approach is based on the idea of pairing sources and destinations. Let's say X is a destination for a call. In this approach, they are going to write a constant, say k_x, just before X. Every time when somebody is going to call X, you will do compare (X-4) and k_x. That is, you are going to check the location 4 byte prior to your control flow transfer with k_x. This is of course makes no sense to check at runtime if you already know what the value of X at compile time or instrumentation time, then you can check it right away. But if you don't know the value of X at compile time, then the check has to be performed at run time. This is of course a very simple and efficient technique. There is no complicated data structure involved. You insert constants in the binary. As it is binary, it is already read only.

Though the mechanism is simple, but there are some difficulties in using it.

The first problem is that if I don't know X statically then how can I generate k_x. So what I will have to do is that I have to see what the set of all possible values it can have. If I can estimate that, then I am going to give the same constant value to all those functions. Let's say X can point to f, g or h functions. If I know that those are the only possible values for X, then I am going to associate the same constant with all three functions. And that is the constant that will be used in the compare. This will immediately lose some precision. This means, any where a call to f is accepted, a call g or h will also have to be accepted.

The problem can be worse. Let's say there is another place where there is an indirect call using Y, where Y can be g, i, or j. So if I have to put one constant for all three g, i, j and one constant for all three f, g, h, then that means I have to use single constants for all the five functions. That means the approach starts losing precision reasonably quickly. Then if you use the same technique for the returns, it will become even harder. So there is a precision loss though in terms of implementation it is relatively simple. Again there are also some cases you will have no idea what these function pointer values will be.

Binary Instrumentation

Program Transformation Techniques

Dynamic analysis works by tracking properties (e.g., taint value) at run-time. How is this accomplished? Via program transformation and instrumentation.

Transformation techniques: two basic approaches:

- source-code based
    - drawbacks:
        - you need source code
        - even if you have source code for an application, you may not have it for all libraries which makes your instrumentation incomplete
    - benefits:
        - higher-level information can lead to more accurate instrumentation and faster instrumentation
- binary based
    - drawbacks: less accurate, slower
    - benefits: don't need source code, no problem w/ libraries

Frequently, source code is not available. Additionally, there are instances where binary code may be simpler (binary code has a smaller set of primitives) – this will depend on what you're trying to track. Another benefit is that if you're developing a binary technique, you get more "bang for your buck" with the binary analysis since it works over a larger set of languages. It is also aimed at users who have no other option (for example a sys admin).

If your clientele is developers, source code static analysis is what you want. If it's an end-user, you want dynamic binary technique.

Binary Analysis And Transformations

Motivation: *[slide 1]*

- No source code needed, language neutral
- Largely os independent
- Ideally, would provide instruction set independent abstractions (far from today's reality).

Two basic approches: *[slide 2]*

Static analysis/transformation

1. binary files are analyzed/transformed

2. benefits

    c. no runtime performance impact

    d. no need for runtime infrastructure

5. weaknesses

    f. prone to error, problem with checksums/signed code

Dynamic analysis/transformation

- code analyzed transformed at runtime
- benefit: more robust accurate
- weakness
    - some runtime overhead
    - runtime infrastructure needed

Previous works: *[slides 3, 4, 5]*

In the '90's most researchers thought that RISC, with its simpler instruction set where all instructions are the same length, was the way to go (SPARC)…most computer security researchers were interested in RISC instruction set. However, all RISC manufacturers went out of business, and X86 is the only game in town. This makes binary analysis more difficult, since instructions have variables lengths.

(see slides for names of various static and dynamic techniques)

Phases in Static Analysis of Binaries: *[slide 6]*

- Disassembly
    - decode instructions
- instruction decoding / understanding
    - analyze to understand what it being done (calling function, returning from function, etc).
- insertion of new code
    - Insert our instrumentation

Disassembly *[slide 7]*

Linear sweep starts at some point, and linearly process instructions from start to end. This is a simple technique to understand, but what tends to happen is that not everything in your executable is code so this process may lead you astray. The other problem is that sometimes functions are aligned on boundaries and there may be some unused bytes between one function's start and another function's end that may lead to further confusion.

Recursive traversal tries to resolve problems with linear sweep. Starts as a linear sweep, but when it encounters a jump or call, etc. stops the linear sweep and continues at the target of the jump or call.

Disassembly impediments *[slide 8]*

Code/data distinctions – it's hard for the analyzer to tell what is code and what is data.

Variable x86 instruction sizes

Indirect Branches:

- mainly due to function pointers, which can be found in:
    - gui code (event handlers)
    - c++ code (virtual functions)
    - others
- functions without explicit call

PIC (Position-independent code)

Hand-coded assembly which may not follow all conventions of the ABI

There may be large sections of code that are reachable only by using function pointers whose values are known only are runtime.

PIC may call itself by using a relative address in order to determine its location…what it might do is to call itself, which will push the return address onto the stack (the current PC) and then simply pop the return value off of the stack.

Disassembling in practice *[slide 9]*

You may incorporate knowledge about the compiler (how it generates code).

You may make some assumptions about which compiler generated the code and what instructions it used to embed knowledge about the way that PIC is used (this is called "idiomatic disassembly").

Another approach is to use a search. First you do traditional disassembly. This may wind up producing some conflicting information. You then use a technique called "speculative disassembly" together with some analysis to decide whether the disassembly is likely to be correct. This is more or less what people use now. The hope is that this is enough to deal with many common problems, but not all

Code Transformation *[slide 10]*

Suppose these challenges of disassembly can be solved (this is true for a large number of binaries, more reliably for executables than for libraries, more reliably for compiled code over hand-generated code, more reliably on Linux than on windows).

Traditional instrumentation means you add instructions. If you have indirect calls, this means that the targets of function pointers may have moved. Since you cannot know the value that will be used in an indirect call, you cannot move anything. Instead, you have to make a new copy of the code, and this new code is where the instrumentation happens. You have to make sure that the target of any control flow transfer (function start, etc) is replaced with a jump to the copy.

Note: Sometimes there isn't enough space, and you have to worry about that, too (if function has one instruction, and then returns for example). A jump takes 5-6 bytes. Mostly these situations are uncommon, but the basic idea is you can't go in and blindly overwrite function bodies.

Static Code Transformation Limitations:

[1] **Code insertion at arbitrary points is very difficult** – The difficulty is that whenever you want to insert code, you need to move the code that follows it. If there is a jump to an address, say at address 00 there is some instruction that is 2 bytes and at 02 there is another instruction. Suppose you want to instrument this instruction which produces an instruction of 6 bytes instead of 2 bytes, so the address 02 is now an invalid address for jump. If the only way to jump to 02 is through direct jumps, you can examine the code and fix the references but there could be indirect jumps which makes it difficult. There are tools that specialize in inserting code only at the beginning and end of a function. Perhaps you know the structure of the beginning and end of the function and that helps you find the space.

Side Discussion: There is a function f and you want to rewrite the body of f, say the rewritten function is f'. You need to find entry points into the code. If there is only one entry point in the code, all you need to do is to insert a jump to the beginning of the function. In this case, we have a new version of the code, the original code has only some portions of the memory space used and the rest is never going to be accessed. It does not really help though. If you think of it as virtual memory and not physical memory, then you need to think in terms of pages being used. If this code is going to be one entire page, then you are just using virtual memory and not physical memory. But chances are that most functions are smaller than a page, in terms of physical memory used you are going to be using double the amount of space or more.

[2] **You cannot see direct correspondence between your program and binary code**.

- A loop may be unrolled.
- **Switch statements translated to jump tables:** If there are other jump points into the code other than the function beginning, then you need to find the other jump points into it. A reason for such indirect jumps could be e.g. Switch statements. If there is a switch statements with many cases. One obvious way to implement it is using if-then-else. This can be very inefficient as you may end up sequentially checking all the values. Another way can be to have a binary search tree, where you will end up having 5 comparisons if there are 32 cases. Another alternative is to generate a jump table. It contain a pointer to the body of the code corresponding to each case. You generate the body of each case and put their addresses into the table and turn the switch statement into one that takes the value of x, indexes the table for value of x into the table and makes the corresponding jump. This is the example of an indirect jump. This is the primary reason why you have indirect jumps.

  Second reason which is not common in programs as much as hand written assembly code or low level libraries is that could have functions that have multiple entry points. You write a program in a high level language that will generate a clean code that has a function which will have a single entry and a single exit. But if you write it in assembly, you will need to have functions that have multiple entry points. This is another reason why you could have multiple entry points into a function.

- **Function arguments may not be explicitly pushed (nor return value popped**): Another issue in binary code is function arguments. You need to figure out how many arguments does a function have to take. So far, we said that functions arguments are passed on the stack, especially if you have a instruction set like x86, the arguments are passed on the stack. There are push statements to push the arguments on the stack. That way you can recognize how many arguments were passed to the function but for optimized code, this is not necessarily true.

Optimized Code Example:

```
#include <stdio.h>

void f(int c) {
        printf("%d\n", c);
}
void h(int i) {
        f(i+1);
}
int i(int j) {
        return j+1;
}
int main(int argc, char*argv[]) {
        h(i(argc));
        f(argc+2);
}
```

```
void f(int c)  {
        printf("%d\n", c);
}
```

Assembly Code: Compile with –S option

Function prologue:

```
pushl %ebp
movl %esp, %ebp
subl $16, %esp

pushl 8(%ebp)
pushl $.LC0 (“%d”)
```

```
void h(int i) {
        f(i+1);
}
```

Assembly Code:

Function prologue:

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
```

No push of arguments

```
incl 8(%ebp)
```

```
call printf

                                    No push of arguments to f, tail call
Function epilogue:                  leave
leave                               jmp f
ret
```

The function f has a prologue and an epilogue. We are pushing the current value of the base pointer, so that is the saved base pointer. We are setting the base pointer to the current value of the stack pointer, so we are setting up the stack frame for the current routine. And then, we allocate space for the local variables and temporaries. These three lines correspond to the printf() call. What's happening is that EBP + 8 is the 'c' argument. EBP + 8 contains the parameter. f takes only one parameter which is c. So this statement is accessing that parameter that is passed into f. It first pushes c onto the stack and then the format string. The format string has been defined as a constant somewhere so it is pushed as an address of the constant. Then it calls printf().

For the function h, the prologue is the same. There is no call, it has been turned into a jump, the reason is that the last operation you do is a call to f. So why call and return if you can jump into f and f will return to the caller of h. So this is called the **tail call optimization.** Call happens to be the last instruction or can be made the last instruction. The compiler figures out whether it can be put as the last instruction. Second thing to notice is that the parameters are not being pushed. It figures that the parameter that it wants is already in the right place. i is the parameter to h and what f is being passed is the same value after doing some arithmetic. Essentially, if I just jump to the beginning of f with stack looking exactly the same way it looked when h was called, then everything would be fine because the parameters are in the same position. Once it figures it out, it just increments the argument. If i is subsequently used after this call, then this cannot be done. It uses some analysis to figure out that after the call there is no use of i.

You can see that you can't always rely on calls to be present and that in general you should assume that you won't know how many parameters are taken by the function. You can't figure out how many parameters does f take. If you analyze the body of the function f, you can figure out how many parameters it uses.

```
main:

pushl   %ebp
movl    %esp, %ebp
pushl   %ebx
subl            $16, %esp
movl    8(%ebp), %ebx
pushl   %ebx
call    i
```

```
movl    %eax, (%esp)  -- Return value in eax register, no argument push
call    h
addl        $2, %ebx
movl    %ebx, 8(%ebp) – No push of arguments to f, tail call
addl        $16, %esp
movl    -4(%ebp), %ebx
leave
jmp     f
```

For the function main(), first 2 lines are same as before. The third line pushes register EBX. The ABI – Application Binary Interface, is something that is architecture and operating system specific, which tells how registers are to be used between callers and callees. You need standardization - otherwise code written in different languages, code generated by multiple compilers cannot work with each other. So the ABI among other things tells you how the stack is to be used, how the parameters, are to be passed, how the return value is handled, what registers can a called function modify and what registers the callee is supposed to preserve. The EBX on x86 and Unix is a register that is supposed to be preserved by the caller. So if the callee needs to use EBX, it needs to store it, use it and then restore it. That's the reason for push EBX here and the instruction which restores EBX. It is not popping because the stack pointer is not at the right place for it to pop. It knows that what location on the stack is ebx at, it just loads it instead of using pop. So, even when parameters are actually passed in, it's not using the existing parameters on the stack. You have to actually put it on the stack; you might not see a push instruction specifically. You might just see an instruction which writes using EBP or ESP.

movl %ebx, 8(%ebp) moves the first argument of main . *8(%ebp)* always gives the first parameter. That's moved to ebx and then ebx is pushed onto the stack which means it becomes the argument to the call of i. And then, the ABI also says that the return value is supposed to go into the eax register. That's why *movl %eax, (%esp)* is moving eax to esp that is basically the top of stack. It is doing this because you want the return value to be passed as parameter to h, so it pushes that. *call h* is calling h and then the next one is the call to f where it has to add.

Notice that it did some sort of analysis to figure out that the parameter argc was moved to ebx and if the callee is supposed to save ebx, it can assume that ebx still has the same value. And therefore you can add 2 to that and then you can see that it doesn't push, just stuffs the arguments onto the stack somehow and then wants to call f but figures that it can use the table.

ABI does not say a whole lot about how local variables are to be stored. The ABI has to worry about caller-callee issues, which means how the parameters are to be passed in etc. Typically when we talk about using shadow stacks and so on, usually that part is not changed.

We talked about the issues in code transformation. Basically all these issues dealt with relocation of code. We said that there is a reasonable way to do code relocation which is for direct calls. I know where is it going to go in the original code and I am going to directly replace it so that it goes to the new location. For indirect calls and indirect jumps, I am going to let it go to its original location where I put a jump to the new location. If we have data that we want to relocate, then in general that is quite hard because if u want to relocate static data, in the assembly code or binary code what you see is something like

mov $0x103236, %ebx

There is no way to figure out if the constant is supposed to be an address or an integer value. So if it knew that it was the address of a static variable, then you can say that I moved the static variable somewhere so I have to adjust this. Since you don't know that, it is impossible to change it because if you change it and it was just a integer constant and you change it arbitrarily, it won't work. This is one of the issues with relocation of static data and this becomes a problem to some extent because if you have position independent code, we said that it computes its data addresses from code. So, if we move that code to a new location, then it might start looking for its data at the new location rather than the old location and typically you don't want to relocate data.

Stack relocation is not difficult and heap relocation is also not difficult. Many of the memory error defenses we have talked about implicitly do stack relocation.

DYNAMIC TRANSFORMATION: The idea is that rather than changing the binary, you are going to change code as it executes. One of the first papers that looked into this is a LibVerify.

LibVerify: It was trying to insert a stack smashing defense in binary code. It made some assumptions. The steps are as follows:

[1] They created a copy of each function on the heap. We can't instrument in place so we need copies. What they are working on is basically binary images in memory. When the process that is running, loads up, that's when they copy the binary image into the heap. So, the assumption that they are making next is that they know the function beginnings and end. Suppose you know that then you can insert your stack protection instrumentation at the beginning and end of the function. So maybe you have to put a canary, and use additional instructions to store a canary onto the stack at the beginning of the function and when you return, you check the canary.

[2] Then what they did was, simply replaced everything in the original function with a trap instruction(1 byte)so if there is any indirect control flow transfer that goes back to the original code, that will immediately cause an exception and in Unix you will get a signal. The signal handler will examine what happened. It is a trap instruction so the signal handler would ask the system what code was executing and it can tell you which instruction was getting executed. Then you can say that if this was the original jump location in the original code, what I should have done is that jump at this new location. And then you can catch that so that the program jumps to the new location.

Basically it replaces the trap with a jump instruction to the corresponding location. In case you don't have enough space, you just leave the trap instruction and you will get a signal every time. So, you are going to handle the signal and jump over there which is going to be very expensive. Overhead of a signal is a context switch type of overhead which is very expensive.

The important thing about this technique is that you can deal with the situation where you did not know anything about the function because the entire original code has been replaced with a trap, and if you somehow didn't know that some piece of that was code and you didn't transform it then you did the copy and instrument, it dint recognize that there was a function there and you recognize this at run time.

DynInst:

It is a pretty robust system. It has been developed over many years. The drawback is that it is not a general purpose instrumentation system. It basically allows to instrument function beginning and end and gives you only a limited set of instrumentation capabilities but tries to do that in a platform independent way.

DynamoRio

This is one of the true dynamic translation techniques where the code is disassembled and instrumented at run time. That's the basic idea of dynamic translation. The idea is that you get the **flexibility of emulation techniques plus the speed of static transformation.**

If you think of emulating binaries, none of the issues that were talked about so far were a problem. If you want to write an emulator that does taint tracking on binaries. You are going to incorporate the semantics of taint tracking in your emulator. You don't have to instrument the binary itself. You don't have to worry about things like the inability to statically disassemble because you are an emulator, you don't have to do anything statically. So, the next instruction that you are supposed to execute, you disassemble that. You figure out what it is supposed to do, you execute it and pick up the next instruction and so on. In the emulator model, there is no reason to disassemble things statically. So, using emulator, binary instrumentation is very easy to do. It gives a lot of flexibility. But, a few of the emulation based techniques are very slow. So you want to look for a faster technique that approaches the speed of static compile time instrumentation. That is what these techniques try to achieve.

Basic idea is JIT translation of basic blocks. In the context of a compiler a basic block is a sequence of instructions that have a single entry point and a single exit point and there is no way to jump in the middle, for e.g. The body of a loop. In this context, a basic block is a sequential sequence of instructions that follow each other. There is no control flow transfer in between.

If you are at the beginning of a basic block, then you can disassemble the entire basic block without any problem because the problem comes up only when you are unable to figure out all the entry points. If you delay the disassembly until you get to the beginning of a basic block, then you know your entry point and you can sequentially disassemble until you hit some jump instruction. So then you don't know where it is going to go especially if it is an indirect jump. So the idea is that you get to examine basic blocks at a time and then you instrument a basic block. The instrumentation basically involves translating the code. From the original code you generate instrumented code. What you are able to do from then on is that you are able to run the instrumented code natively so that each basic block incurs the translation overhead only once. After that you translate it and the translated version gets executed and it becomes faster. So, the startup is going to be slow, but once the program starts up and goes through most of its code, after that it is going to be quite fast.

Address management: Data addresses are not changed. Code addresses are known at run time. Code segments are the ones that are going to be put at arbitrary places. There is original code and transformed code. As it does the transformation, it can maintain a mapping of original locations to the new locations. Then, there are indirect jumps and calls. If there is a direct jump then the target of the jump is known at the time the dynamic translation is done. In that case, let's say there is the original basic block which has a jump back to the beginning, and then the instrumented basic block will have a jump back to its beginning. Most jumps will not have additional overheads after the translation but you could have indirect jumps and calls. In those cases there is no way to statically fix it because the indirect jumps and calls are going to go to the original location of the code, so it has to do a dynamic translation of those addresses. It has some data structure which has the original address and the new address as it does this translation and based on that translation table, whenever it sees an indirect jump or call it is going to go and pick up the new address and jump there which means that indirect calls and jump instructions always have to be emulated. Whenever there is an indirect call or jump, it needs to stop executing in native mode, jump into the emulator – the emulator figures out the address to which you need to jump and then comes back to the regular mode. This is called a context switch in this terminology because the emulator is a program and the emulator is going to use registers for its own computation and the translated program is also a program which is going to use the registers for its own purposes. Every time you switch between the two, you have to save the registers that the emulator needs and switch back. Context switches are expensive but saving registers is much cheaper than jumping into the operating system and jumping back.

It basically tries to stay in instrumented code as long as possible to avoid context switches. It has an interesting effect. Let's say your code has a function that is entered and from there we go to another function and come back and then execute. Control flow that is statically predictable, instead of making you jump all over the places, DynamoRio can put the code together as one long sequence of instructions. The aim to avoid jump is code locality. If you assemble the code that will execute sequentially together, then you will get

better code locality and improved performance. This is called a trace, something that executes sequentially even if there is a jump. If the jump or call is to a known location, then they can just remove it and inline the whole thing.

Performance Benchmarks:

Unix: These are the standard benchmarks. 1.0 represents the native run time of the program. RIO is the performance with just the dynamic translation going on without adding any instrumentation. It is NULL instrumentation. You have the overhead of indirect branches but there is no other overhead. The blue bar which reflects the RIO is pretty much at one. For some programs it speeds up. They get better locality due to trace cache. Programs like Applu have better performance. These benchmarks are CPU intensive benchmarks. With benchmarks, if you run the program long enough, you can reduce the overheads to quite a low value. So, the only thing you should expect is that only the indirect control flow transfers are going to have any effect. You will see very different performance if you run these programs during the startup phase. It will be significantly slower.

Windows: On Windows they don't do very well. Whenever they do dynamic code generation, they need to change privilege on the dynamically generated code pages because you want to make sure that the program cannot modify itself. This means that, when the code is generated, that should be writable memory page. After the code has been generated, it should be write protected. Everytime it generates a basic block, it needs to make sure that the protection bits are set. For this it needs to make additional system calls and systems calls in windows tend to be more expensive and so they are slower on windows.

Program Shepherding: An application that was built using this framework. They wanted to enforce a bunch of policies. The reason for the term shepherding is that it is hard to figure out what they were doing. They were basically trying to block the major security threats. They were trying to deal with code injection and existing code attacks. The original program should not be able to dynamically generate code and execute it at run time.

The last point is that, it should not be possible for the code to compromise the DynamoRio framework. They added the checks for enforcing code origin and code control flow type of policies and that adds to the overhead. In the performance chart, the white one (RIO + protection + program shepherding), the protection makes sure that a malicious program cannot circumvent the DynamoRio framework. It does not make sense to talk abt DynamoRio + Shepherding in absence of protection.

DynamoRio API Overview: DynamoRio exposes API's and some examples that explain how to write the instrumentation things using DynamoRio. The idea is that if you want to write an instrumentation, you don't want to deal with how to disassemble and all sorts of low level issues. It provides a convenient API that allows you to do all of this.

These techniques are quite robust and can work with a large number of applications. DynamoRio certainly works on all kinds of windows executables and DLLs. It will not work for all programs but for some. These types of dynamic instrumentation techniques are quite robust and can work with arbitrary programs. It makes some assumptions and it is unlikely that the assumptions don't hold.

Position Independent Code basically looks at the code address and generates the data address from the code address. They dont want to change those locations so consider the code:

```
        call    rel + 1
+1      pop     ebx
        mov     20(%ebx), ecx
```

These three instructions can be used to move some data used in a PIC library. If you move this code to a new location and execute it, it would not work.

The transparency issue here is that the translated code should work exactly like the code before translation. This is one case where the translation introduces a problem. To deal with this, when calls are done, they don't push – they change the call instruction. Instead of pushing the address of the calling instruction, they push the address of the original call instruction. That way even though the code is moved, what is on the stack is the old location. Then they use a second stack for their own purpose. Whenever a return comes, they do the return from the second stack which has the address corresponding to the new code location. Their approach is that any data or register that is accessible by the code should not be changed. Those data values should not be changed. The fact that it is being executed this way is not going to be visible.

Thinking of another application, there is some piece of code which is going to examine itself and is going to dynamically checksum itself. Only if the checksum works out, it will run otherwise it will abort. Some kind of tamper detection is built into the code. Because of the way they do this, it is going to work fine. The point is that anytime you take the address of some function, it can be later used as data. Whenever the code tries to examine itself, it is going to examine the original version and not the translated version. These are the reasons that the tool is robust. It is able to deal with all kinds of strange behavior in certain types of code.

Other Dynamic Translation Tools:

[1] **Pin** is a tool developed largely in the context of Linux.

[2] **Strata** is platform neutral but is less mature

[3] **Valgrind** – **memcheck** version of valgrind is used to check run time memory errors. It is not as powerful as source code based memory checkers but it gives you quite a few functionalities.

[4] **Qemu** incorproates dynamic code generation. Another point is that it is a whole system emulation tool. You can emulate operating system and applications in one shot. They talk about it as some sort of a virtual machine type of environment. Entire machine is emulated.

The downside of these tools is their poor performance. Once you start doing complex instrumentation then the technique is slow.