Binary Instrumentation

<u>Program Transformation Techniques</u>

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:

1. source-code based
   a. drawbacks:
      i. you need source code
      ii. even if you have source code for an application, you may not have it for all libraries which makes your instrumentation incomplete
   b. benefits:
      i. higher-level information can lead to more accurate instrumentation and faster instrumentation
2. binary based
   a. drawbacks: less accurate, slower
   b. 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.

<u>Binary Analysis And Transformations</u>

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

- binary files are analyzed/transformed
- benefits
  - no runtime performance impact
  - no need for runtime infrastructure
- weaknesses
  - 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]*

1. Disassembly

    a. decode instructions
2. instruction decoding / understanding
    a. analyze to understand what it being done (calling function, returning from function, etc).
3. insertion of new code
    a. 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**.

(i) A loop may be unrolled.

(ii) **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.

(iii) **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”)
call printf

Function epilogue:
leave
ret
```

```
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)

No push of arguments to f, tail call
leave
jmp f
```

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.