

*ANALYSIS TOOLS*

# **ATTACK**

*CONSTRUCTION KIT*



**metrowerks®**  
*Software Starts Here* ◀



# Analysis Tools Construction Kit

---

|   |                     |
|---|---------------------|
| <b><i>Lesson 01: Introduction and Overview</i></b>    | <b><i>01-1</i></b>  |
| Why ATtaCK?   | 01-1                |
| Course Materials                                      | 01-2                |
| Analyzing Applications                                | 01-4                |
| The Analysis Tools Construction Kit                   | 01-6                |
| The ATtaCK Process                                    | 01-7                |
| Developing ATtaCK Tools                               | 01-10               |
| Creating Stationery                                   | 01-11               |
| BareBones: The World's Least Useful ATtaCK Tool       | 01-18               |
| Lesson 01 Quiz  | 01-22               |
| <b><i>Lesson 02: Examining an Application</i></b>     | <b><i>02-23</i></b> |
| Anatomy of an Application                             | 02-23               |
| ATtaCK Types: Objects                                 | 02-26               |
| ATtaCK Types: Data Types                              | 02-29               |
| Startup and Shutdown                                  | 02-31               |
| Navigating with Iterators                             | 02-33               |
| Finding Code Structures                               | 02-36               |
| Querying Code Attributes                              | 02-40               |
| A Static Analysis Tool                                | 02-47               |
| Lesson 02 Assignment                                  | 02-53               |
| Lesson 02 Quiz  | 02-54               |
| <b><i>Lesson 03: Instrumenting an Application</i></b> | <b><i>03-56</i></b> |
| Instrumentation Concepts                              | 03-56               |
| Instrumentation Calls                                 | 03-58               |
| Instrumenting an Application                          | 03-60               |
| Instrumentation Sequencing                            | 03-63               |
| Instrumentation Sequencing—Instructions               | 03-65               |
| Creating Analysis Code                                | 03-67               |
| Declaring Analysis Code                               | 03-69               |
| A Minimal ATtaCK Tool                                 | 03-71               |

|   |                      |
|---|----------------------|
| <b>ProcCount: Navigation and Instrumentation</b>              | <b>03-72</b>         |
| <b>ProcCount: Analysis and Output</b>                         | <b>03-74</b>         |
| <b>Lesson 03 Assignment</b>                                   | <b>03-76</b>         |
| <b>Lesson 03 Quiz</b>   | <b>03-78</b>         |
| <b><i>Lesson 04: Running and Analyzing an Application</i></b> | <b><i>04-79</i></b>  |
| <b>Execution Concepts</b>                                     | <b>04-79</b>         |
| <b>Executing an Application</b>                               | <b>04-81</b>         |
| <b>Configuration Objects</b>                                  | <b>04-83</b>         |
| <b>ATtACK Events</b>  | <b>04-86</b>         |
| <b>Controlling the Application</b>                            | <b>04-88</b>         |
| <b>Critical Sections</b>                                      | <b>04-90</b>         |
| <b>Communicating with the Application</b>                     | <b>04-93</b>         |
| <b>Arrays and Structures</b>                                  | <b>04-94</b>         |
| <b>A Simple Profiler</b>                                      | <b>04-99</b>         |
| <b>SimpProf: Navigation</b>                                   | <b>04-100</b>        |
| <b>SimpProf: Instrumentation and Analysis</b>                 | <b>04-101</b>        |
| <b>SimpProf: Execution and Output</b>                         | <b>04-103</b>        |
| <b>Lesson 04 Assignment</b>                                   | <b>04-105</b>        |
| <b>Lesson 04 Quiz</b>   | <b>04-107</b>        |
| <b><i>Lesson 05: Designing Analysis Tools</i></b>             | <b><i>05-109</i></b> |
| <b>The Road Ahead</b>   | <b>05-109</b>        |
| <b>Principles of Code Analysis</b>                            | <b>05-109</b>        |
| <b>Designing Analysis Tools</b>                               | <b>05-112</b>        |
| <b>Dynamic Arguments</b>                                      | <b>05-114</b>        |
| <b>Working with Registers</b>                                 | <b>05-118</b>        |
| <b>Measuring Performance Counter Events</b>                   | <b>05-121</b>        |
| <b>PS2Counter: Analysis Code</b>                              | <b>05-124</b>        |
| <b>PS2Counter: Instrumentation</b>                            | <b>05-127</b>        |
| <b>PS2Counter: Navigation, Execution and Output</b>           | <b>05-129</b>        |
| <b>Lesson 05 Assignment</b>                                   | <b>05-131</b>        |
| <b>Lesson 05 Quiz</b>   | <b>05-133</b>        |
| <b><i>Lesson 06: Profiling Applications</i></b>               | <b><i>06-135</i></b> |

|   |                      |
|---|----------------------|
| <b>Verifying Code Compliance</b>                          | <b>06-135</b>        |
| <b>TRC: Initialization and Navigation</b>                 | <b>06-136</b>        |
| <b>TRC: Analysis and Reporting</b>                        | <b>06-139</b>        |
| <b>Analyzing Branch Prediction</b>                        | <b>06-141</b>        |
| <b>Branch: Improvement</b>                                | <b>06-143</b>        |
| <b>BranchPred: Analysis</b>                               | <b>06-146</b>        |
| <b>Looking for Inlining Opportunities</b>                 | <b>06-149</b>        |
| <b>Inliner: Instrumenation and Analysis</b>               | <b>06-151</b>        |
| <b>Inliner: Output, Interpretation and Improvements</b>   | <b>06-154</b>        |
| <b>Detecting Load Delays</b>                              | <b>06-157</b>        |
| <b><i>Lesson 07: Analyzing Memory and Cache Usage</i></b> | <b><i>07-162</i></b> |
| <b>Catching Misaligned Memory Accesses</b>                | <b>07-162</b>        |
| <b>Hunting Down Memory Leaks</b>                          | <b>07-164</b>        |
| <b>Plumber: Execution and Output</b>                      | <b>07-167</b>        |
| <b>Monitoring Stack Depth</b>                             | <b>07-170</b>        |
| <b>Analyzing Cache Usage</b>                              | <b>07-174</b>        |
| <b>PS2Cache</b>   | <b>07-176</b>        |
| <b><i>How To Contact Metrowerks</i></b>                   | <b><i>07-181</i></b> |
| <b><i>Quiz Answer Key</i></b>                             | <b><i>07-182</i></b> |

## Lesson 01: Introduction and Overview

---

### *Why ATtaCK?*

#### Why You Need This Course

Metrowerks' Analysis Tools Construction Kit, ATtaCK, is a powerful new framework for developing custom profiling and debugging tools. The robust API enables you to develop anything from a disassembler to a cache simulator, while the simple programming model lets you write quick "throw-away" tools tailored to answer specific questions. This course will cover both ends of that spectrum, teaching you how to write sophisticated, general-purpose analyzers as well as simple, one-off tools.

You should take this course if you are planning on using ATtaCK to analyze your PlayStation®2 application's performance. You will get the most out of this course if you have experience using profiling tools on other platforms, plus an extensive knowledge of EE assembly programming. However, part of the power of ATtaCK is its simplicity—you don't have to be an expert to use it. As long as you're comfortable with C programming, you can learn how to create ATtaCK tools.

There are seven lessons in this course. The first lesson introduces the concepts behind ATtaCK, walks you through its installation and configuration, and generally gives you the "big picture." The next four lessons cover the complete ATtaCK API, from opening an application all the way to running it on the target system. The final two lessons are labs, each examining four "real-world" analysis tools for you to adopt, adapt and improve.

#### How to Take This Course

In an ideal world, people are constantly profiling and optimizing their applications as they program. In the real world, most people don't bother to profile their applications until they hit a performance problem, at which point they need answers right away. If you're in that situation, take heart: You're the exact person I've designed this course for.

By dedicating yourself full-time to the subject, you can become proficient in ATtaCK in just a few days, and can master it within a week. In a classroom environment with a live teacher, each lesson should take half a day. Online classes tend to go at a slower pace, but even then, you can cover all seven lessons within a single work-week without quite abandoning all your other responsibilities.

And trust me, that week will be well spent. I strongly recommend doing all the classes in this course in sequence, in as short a timeframe as possible.

Everything you need to learn is in this course. Reading the documentation as well is certainly a good idea, but it's not necessary. There are, of course, a few places where I cop out and point you to the docs, rather than copy six pages of tables into the course. But

for the most part, you'll only need to crack the manual for obscure details, precise definitions or other background information.

## **Prerequisites**

Obviously, to get the full benefit from this course you'll need the ATtaCK Framework for PlayStation 2. That in turn requires Metrowerks CodeWarrior Professional for Windows/x86 (R6 or higher) and Metrowerks CodeWarrior for Sony PlayStation 2 (R2.5 or higher). ATtaCK lets you get a lot of information out of your application without ever actually running the program, but if you want to try all the sample programs in the course you'll need access to a T10000.

To take this course, you should:

- Be comfortable with C programming
- Be familiar with the use of the CodeWarrior IDE
- Be familiar with PS2 development

To get the maximum benefit out of the course, you should also:

- Be comfortable with analysis tools on other platforms, such as VTune for Intel x86
- Be familiar with EE assembly language
- Have a PS2 application that you want to profile

## ***Course Materials***

### **Lessons**

If you're reading this, you obviously know where the lessons are. You can proceed at your own pace through the course—you've got full access to all the lessons. If you skip ahead, though, be prepared to back up and review, because I generally only cover a subject once!

### **Examples**

After the lessons themselves, the examples are the most important part of this course. The example programs are all usable as tools in their own right, and they make great starting points for building your own custom tools.

Many of the example programs are included as part of your ATtaCK installation. (If you haven't installed ATtaCK yet, don't worry—that's coming later in this lesson.) The other examples can be found in the "supplemental material" folder.

## Documentation

Online documentation should be part of your ATtaCK distribution. However, this course is designed to give you a basic understanding of ATtaCK for the PlayStation 2 without reading the documentation at all. We'll cover every function in the entire API, but not in the level of detail the docs give—instead, we'll just look at how the functions are *usually* used. If you find yourself working on advanced tools later on, you'll probably want to read the docs to pick up the subtleties I'm skipping.

Also, note that this course is written specifically for PlayStation 2. When ATtaCK becomes available for another platform, the course will most likely be adapted to that platform. If not, then you can still use this course—most of ATtaCK is platform-independent, after all. Just be sure to read the platform-specific API docs alongside the lessons, so that you can see where they diverge.

## Quizzes

The first five lessons include interactive multiple-choice tests at the end. These are "graded," in the sense that there are right answers you're expected to give. Aim for a perfect score—if you get a question wrong, it probably indicates a topic you should look at again.

## Exercises

Unlike quizzes, the exercises aren't graded. They're sample problems, real-world (or at least realistic) situations you might face. As such, they don't really have "right" answers. Instead, I explain how *I* would approach the problem, which can give you ideas for your own tools.

Only Lessons 02 through 05 have exercises. There's not much point in having exercises in this lesson—we're just getting started, so you're not ready to tackle real-world problems yet. The last two lessons, on the other hand, are labs, in which we'll examine many sample programs. In a sense, those lessons are nothing *but* exercises!

## Instructor Interaction

Depending on when you take the course, you may have the opportunity for live chats, access to web message boards, email Q and A, and other instructor interaction. If this is available, the main course website will give the schedule and links. If not, though, don't worry—the course has been written to stand alone.

By the way, the person *writing* this course is Stephen Beeman. While I hope to also *teach* this course, the person teaching the course may be someone else entirely. Please don't give him or her a hard time over my lessons!



## Course Conventions

Many lessons have notes, text running in an indented paragraph. Notes are information tangential to the main text, but still useful or at least interesting. You should read the notes after you finish reading the rest of the lesson, or when the text says "see notes."

Oh, and literal code and other text that should be typed as written follows the standard practice of appearing in Courier, like this.

## Analyzing Applications

Analysis is defined as "an examination of a complex, its elements, and their relations." That pretty much sums up the role of analysis in programming: an examination of an application, its functions and their relationships. Specifically, code analysis answers two questions: "Is everything working as expected? If not, why not?" The first question is *detection*. The second is *diagnosis*.

### Detection

The most common method of detection is human testing: simply having a human run the program and see what breaks. A more sophisticated method is regression testing, where a program's output is automatically compared to expected output. However, that is not so easy for game programming: Most problems are aesthetic, and require a playtester to spot.

Relying so much on human-monitored testing, detection often doesn't use any tools. That however, is a mistake: People don't use tools because they're unaware of what's available, because for the longest time there simply weren't good analysis tools available for game programming and especially for game platforms.

However, computer assistance can greatly amplify the power of human testing. In some cases, it can replace the need for testing.

Regression testing we've already mentioned. It's hard to do with games, but still possible. For example, physics routines, AI routines and similar code often produce output that can be predicted (either deterministically or statistically). This allows a regression analysis tool to compare the program's output with the expected output; if they don't match, then you've detected a problem.

Unfortunately, while a powerful technique, this almost always requires custom tools. More to the point, it almost always requires special changes to the application, breaking subsystems out into standalone testbeds and so forth. So we won't cover this subject much in this course, but you should keep it in mind—you might figure out a way to apply the ideas to your code.

More relevant are bounds checkers and memory-leak detectors. In a way, these are regression analysis tools too. Bounds checkers test the app against the expected behavior that arrays bounds won't be overstepped and memory won't be trashed. Leak detectors

test for the expectation that all the memory allocated gets released before the end of the program.

Code validation tools operate the same way. They test the app against the expected behavior that no illegal parameters get passed to functions, or no illegal API routines get called.

One code validation tool that works on source code is called lint. I strongly recommend using this program, which is available in several versions... some freeware, some not. This program looks for source errors, ranging from obvious, such as saying "if (x = 3)" rather than "if (x == 3)", to subtle errors, such as misplaced semicolons and the like.

But code validation can work on compiled code, too. One of the programs we will look at looks for trashed registers. If you write in assembly, this can be handy. If you write in C/C++, you might not see much use for this, but it can catch compiler bugs. I've found three genuine compiler errors in my programming career (*never* in Metrowerks compilers, I must add!), and in each case a tool like this would have saved me days of searching.

Detection tools can also help human testers do their jobs. A code coverage tool like VisualCoverage, for instance, simply checks to see what functions (or what lines within functions) have been executed. This lets the tester know when every part of the app has been exercised, so that he can feel confident he's given the app a thorough test.

## Diagnosis

If detection is playtesting, then diagnosis is debugging: Once we know a problem exists, we track down the cause. Of course, the line between detection and debugging can be fuzzy, because often when we detect a problem we diagnose it at the same time.

For instance, you might put an assert in your code so that, when a function fails, it generates an error message. That's detection. If that error message contained information about what the program was trying to do—that is, if it contained diagnostic information—then the assert would be a diagnostic tool as well.

Diagnosis presents a special challenge with large, complicated programs like games, because you cannot always recreate the exact problem. Detection tools are thus very important: Every time the problem occurs, the tool will give information to the playtester, and (hopefully!) the playtester notes it for you. Diagnostic tools that can be used quickly, or even all the time, are also important—if the playtester can immediately gather some diagnostic information, that makes intermittent problems much easier to find and fix.

But generally, diagnosis is a separate process performed by the programmer with separate tools. A debugger is the most familiar diagnostic tool, for instance.

Another diagnostic tool is a memory allocation tracker. A leak detector just tells you that memory has leaked, or might tell you that an allocation of 16 bytes leaked. An allocation tracker tells you "the memory allocated on line 146 of gameloop.c never gets released." Armed with that kind of info, fixing the problem is usually pretty easy.

A profiler is another diagnostic tool. Speed problems are, after all, just another kind of bug. A good profiler diagnoses these bugs, telling you why—or at least *where*—they’re happening.

This discussion is by no means exhaustive. The point is to get you thinking about tools. Game developers are used to making do with poor tools and, in my experience, need some reeducation as we enter the age of better tools.

## ***The Analysis Tools Construction Kit***

Okay, now that I’ve told you stuff you probably already knew, we’re going to get to the stuff you *don’t* know.

### **What is ATtaCK?**

ATtaCK, the Analysis Tools Construction Kit, is an application framework to create custom analysis tools. The previous page gave you a pretty good idea of what analysis tools are, so now all you need to know is what an application framework is.

A framework is simply a library of code that provides a specific set of related features. If the features supported by the framework include facilities for initializing, quitting and other application-specific activities, you have an application framework.

In this case, the ATtaCK framework provides functions not only for initializing and quitting, but also for loading and examining target applications, defining analysis code, adding instrumentation calls into the target, and controlling a remote development system.

### **What Does ATtaCK Do?**

ATtaCK tools can be used for both detection and diagnosis. They perform analysis two ways, by observing the application run on the target platform and by simulating the application’s behavior.

Running on the target platform means adding analysis code. Asserts are analysis code—code that doesn’t contribute to your program, but simply gathers and reports data. ATtaCK lets you add analysis code without recompiling—you write and compile the code separately, and then use ATtaCK (in a process called instrumentation) to insert that code into the target application.

Simulating the target’s behavior means looking at the code and figuring out what the CPU will do with it. You do this on a trivial level all the time—when you look at your source code and see the line “ $x = x + 1$ ,” you know that the CPU is going to increment the value of  $x$  by 1. ATtaCK does the same thing, just on a bigger scale.

But in all this discussion of what ATtaCK does, the important thing to remember is that ATtaCK is a construction kit—it doesn’t do anything unless you tell it to! To get the most out of ATtaCK, you will have to design and build your own analysis tools. This requires

you to understand not only the mechanics of ATtaCK, but also the principles of code analysis *and* the specifics of your application. In Lesson 05, we'll spend some time talking about this concept.

This is an unusual approach—most analysis tools, including Metrowerks' own CodeWarrior Analysis Tools package—come ready-to-use, with generic interfaces able to handle any application. The ATtaCK approach requires more effort on your part, but it offers greater rewards: ATtaCK can handle any analysis task, even ones unique to your application—and we all know how unique game code can be!

Another advantage is subtle but important. Other analysis tools have an interface that you must learn. The interface for ATtaCK is essentially C, which you already know. You know how to describe sequences of activity in C; ATtaCK does all the work, so that description is all you really need to perform the analysis.

A "ready-to-use" tool with the power and flexibility of ATtaCK would certainly have to incorporate some kind of scripting language to drive its behavior. Once you have to learn a scripting language, why not make that language C? And once you're writing a program in C, why not compile that program into a stand-alone tool that you can give to your programmers *and* testers? The ATtaCK approach truly rewards the effort.

And, truth be told, many analysis tasks are common across all applications, and for some of those tasks we provide you with pre-built tools. ATtaCK comes with five tools itself, and this course includes five more. Best of all, since these tools are built with ATtaCK and we provide you the source code, you can adapt and extend them all you want.

And really, developing ATtaCK tools is not that difficult—a junior programmer can do it, although analysis is important enough that it merits the attention of your team's senior coders. The ATtaCK framework does almost all the work for you. All you have to do is figure out what information you want to collect from your application and write code to do that.

Of course, with a seven-lesson course dedicated to the subject, ATtaCK may not seem simple. But this one course will teach you how to use ATtaCK as a debugger, a profiler, a code-coverage tool, a code validation tool, a memory leak detector, and more. It's much easier than learning six different tools in a week, none of which might be compatible with each other!

## ***The ATtaCK Process***

The typical ATtaCK tool comprises two programs, an instrumentation tool and analysis code. The instrumentation tool is a simple Windows command-line program that opens a target application, examines it, inserts instrumentation calls and runs the application on the target system. The analysis code is a limited PlayStation 2 program that ATtaCK merges with the target application; the instrumentation calls invoke functions in the analysis code to gather data.

Although every ATtaCK tool is different, each one has to perform certain steps in its instrumentation tool and analysis code:

1. **Read in user options.** At a minimum, the tool needs to know the name of the target application. How you accomplish this is completely up to you—you can hard-code them if you feel like it, but better practice is to read them in from the command line.
2. **Initialize an ATtaCK session.** ATtaCK manages all its resources using a session object. All your tool has to do is call one simple function to create and initialize a session; ATtaCK handles the rest invisibly. Initialization is covered in Lesson 02.
3. **Open application.** ATtaCK loads the application in from the disk and prepares it for analysis. If you're going to add instrumentation code, the tool will eventually create a new application; the filename for that new program is specified here at this step. The open function is covered in Lesson 02.
4. **Step through code using iterators.** Most analysis tools will need to process every function in the target application. This is done using iterators, special objects ATtaCK creates to let your tool step through every code element one at a time. Code iteration is perhaps the most important concept in this entire course, so we'll be beating the subject to death in Lesson 02.
5. **Declare analysis routines and data structures.** Analysis routines are functions you write to gather data from the target application at run-time. Your instrumentation tool will use ATtaCK to modify the target application to call these routines, and so you must tell ATtaCK the names and definitions of the routines. Likewise, you must declare to ATtaCK the layout of any data structures that you want to pass back and forth to the analysis code. Analysis routines are covered in Lesson 03; data structures are discussed in Lesson 04.
6. **Add instrumentation calls and/or perform static analysis.** Instrumentation calls are code inserted into the target application to gather data at run-time, and are covered in Lesson 03. Static analysis consists of examining the application code directly, without running it, and is covered in Lesson 02. But really, this step is the entire point of ATtaCK, and one way or another *all* the remaining lessons will cover it!
7. **Write out or close the application.** The instrumentation calls added to the target application create a new version of that application. This is saved to disk, either to run later or into a temporary file to download and run immediately. If you're just doing static analysis, you won't be writing anything; instead you simply close the application. Writing is covered in Lesson 03; closing is covered in Lesson 02.
8. **Download and run instrumented application.** ATtaCK handles all communication with the target system, downloading and launching your instrumented application. You have a small measure of interactive control from the host—you can pause, resume or halt the target—but for the most part you just

- run your instrumented application the way you'd run any other program on the target system. We'll come back to this in Lesson 04.
9. **Analysis code gathers data in instrumented application.** As the instrumented application runs on the target system, your original code executes calls to the analysis code added by ATtaCK. The analysis code performs three "subtasks":
    - 9A *Allocate space for analysis data.* The data your analysis code collects has to go somewhere until the host system is ready to receive it. That somewhere is memory on the target system, which the analysis code must allocate somehow. Usually you allow ATtaCK to handle this for you. We'll look at this subject briefly in Lesson 03, and in more detail in Lesson 04.
    - 9B *Initialize analysis data.* Once you've got a buffer allocated for the data, you need to initialize the data before collection begins. You may also want to re-initialize the data. Again, there are ways to let ATtaCK handle this for you, or you can do it by hand. This is also covered in Lesson 04.
    - 9C *Gather analysis data.* At its simplest, analyses code just increments counters. Since you want to keep your analysis code as lightweight as possible, "simplest" is usually how things wind up being. This is covered in Lesson 03, to the extent that you need me to teach you how to increment a variable...
  10. **Read data from instrumented application.** As the application runs on the target system, the instrumentation code you added captures data and stores it in a buffer on the target. When you've compiled enough data for analysis, you use ATtaCK to read that buffer back into your analysis tool. This is covered in Lesson 04.
  11. **Analyze and output results.** At this point, you've compiled a set of data about the target application, by capturing data from a live application and/or by statically examining the application's machine code. All that remains is for you to analyze, store and display your results. The bad news is, this is something ATtaCK can't help you with. The good news is, hey, that's why you get paid the big bucks. I've done my share of profiling and optimization over my dozen years in the games business, and in Lessons 05 through 07, I'll share some of what I've learned.
  12. **Close ATtaCK.** Being a good citizen of the programming world, you always free what you've allocated, destroy what you've created and otherwise clean up after yourself. Right? ATtaCK makes this as easy as possible and we'll cover it in Lesson 02.

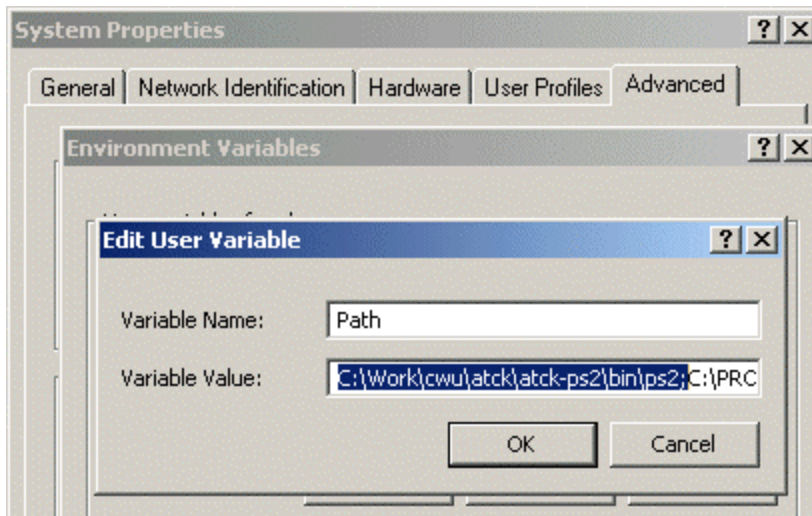
## Developing ATtaCK Tools

### Installing ATtaCK

The first step toward developing ATtaCK tools is, of course, to install it. Whether you received ATtaCK by download or on a CD, the installation package should have come with instructions, so start with those.

Make a note of whatever folder you install ATtaCK into, because you'll need it often through the rest of this lesson. By default, this will be something like `C:\Program Files\Metrowerks\ATtaCK for PS2`. I'll refer to it as "your ATtaCK folder."

Once that's done, you need to make sure that your `PATH` environment variable is set properly. ATtaCK uses this variable to search for `atck.dll`, which is where the ATtaCK libraries live. This file is found in the `bin\ps2` subdirectory of your ATtaCK folder, so you need to add the full path to that subdirectory to your `PATH` variable.



*Figure 01-01: Setting the Search Path on Windows 2000*

On Windows 98/Me, `PATH` is set in `autoexec.bat`. On Windows 2000, this variable is much harder to get to: Right-click on *My Computer* and select *Properties...* Then click on the *Advanced* tab and hit *Environment Variables...* Finally, select the `Path` variable and hit *Edit*.

### Using the CodeWarrior IDE

Each ATtaCK tool is composed of an instrumentation tool and a set of analysis code. The instrumentation tool is a standard Windows application built with any Windows C/C++ compiler (preferably CodeWarrior!), while the analysis code is a limited PlayStation 2 program that must be compiled with CodeWarrior for PlayStation 2.

If you're not familiar with the CodeWarrior for Windows IDE, there's a CodeWarriorU.com course on the subject you should take. Go to <http://www.codewarrioru.com> for more information.

Really, though, if you need ATtaCK, then you've probably already built a PlayStation 2 application using CodeWarrior, so I feel pretty safe in assuming that you know how the IDE works. Besides, it's just an IDE—I'm sure you've dealt with these before.

Nevertheless, there are a number of project settings that have to be "just right" to make ATtaCK work. You might not have ever dealt with some of these settings before, so that is a subject we need to cover. While we're at it, we'll look at project stationery, CodeWarrior's mechanism for setting up new projects with default settings.

This is very important for ATtaCK—because ATtaCK apps, especially the analysis code, have some special features, they require a lot of changes from the normal project defaults. Stationery will let you make those changes just a single time, then duplicate them in all your future projects. That's a tremendous time-saver, so let's go ahead and create ATtaCK project stationery now.

## ***Creating Stationery***

Stationery is just a normal project located in the Stationery folder below your CodeWarrior installation. You create stationery by creating new projects in the stationery folder. Each new project, of course, is itself created with stationery, which we'll specify.

## **Creating the Project**

A single CodeWarrior project can have multiple targets. Normally, each of these targets is just a different configuration of the program for one platform. For example, I'm sure you're familiar with "debug" and "release" builds.

However, CodeWarrior is designed for cross-platform development. Beyond just having different settings, each target can use a different compiler and be built for a different platform. This is tremendously convenient for ATtaCK tool development, because it means our instrumentation tool and analysis code can share the same project.

Open up CodeWarrior. Go to **New...** Select **Empty Project**. Give it a name—ATtaCK Tool will do—and hit **OK**. It doesn't matter where this project is, because once we're finished creating the stationery, we're going to move it.

## **Creating the Folders**

Use Explorer, File Manager or a plain old DOS prompt to go to the new folder containing your project. Create the following subfolders:

- **Inst**, which will hold the instrumentation tool
- **Anal**, which will hold the analysis code



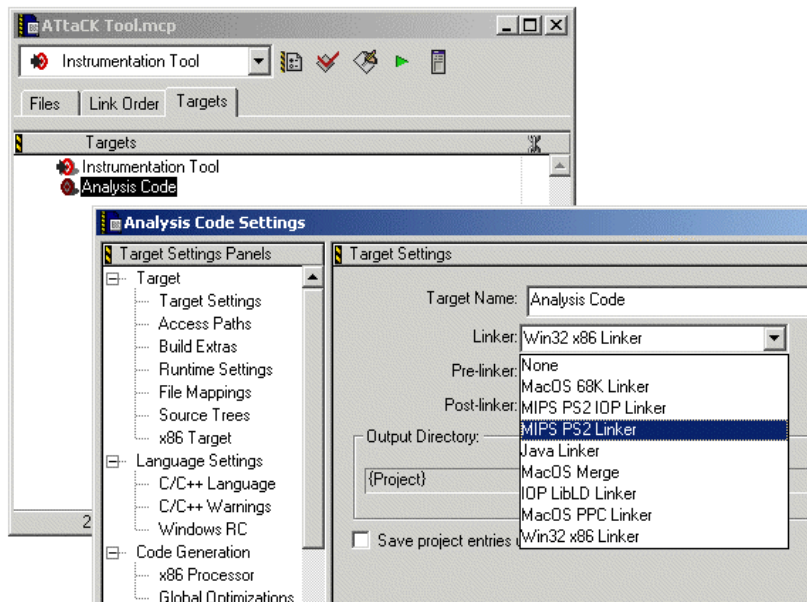
- Shared, which will contain any project-specific include files shared between the instrumentation tool and analysis code
- And Bin, which is where each target's output will be placed

Now go back to CodeWarrior.

## Creating the Targets

All projects must have at least one target. We need two, so we'll have to add one. From the menu, select `Project > Create Target...` In the dialog box that comes up type `Analysis Code` and hit `ENTER`.

Click on the project window's `Targets` tab to see our two targets. Double-click on the original target, the one called "ATtaCK Tool." This brings up the "Target Settings" dialog box, which we'll be seeing a lot of for the next few pages. Right now, change the target name to `Instrumentation Tool`, and make sure the linker is set to `Win32 x86 Linker`. Hit `OK`. Now double-click on the new "Analysis Code" target, change its linker to `MIPS PS2 Linker` and hit `OK`.



**Figure 01-02: Changing the "Analysis Code" Target's Linker**

Which other target settings are available depends entirely on the target's linker. Thus, when you change the "Analysis Code" target from Windows to PlayStation 2, you can see that the settings available change as well. For instance, the "Windows RC" language settings get replaced with settings for the "GNU DVP Assembler" and "MW GAS Assembler."

Now that the two targets have the proper settings available, we need to, well, set them. We'll start with the easier of the two, the "Instrumentation Tool" project.

---

## Using Two Separate CodeWarriors

---

The main text describes the ideal situation, in which your CodeWarrior for Windows and CodeWarrior for PlayStation 2 installations share the same folder. This lets you combine the instrumentation tool and analysis code into a single project.

If you have two separate installations of CodeWarrior, your ATtaCK tools will have to be split into two separate projects, too. You can still follow this lesson's instructions—just put each target in its own project.

---

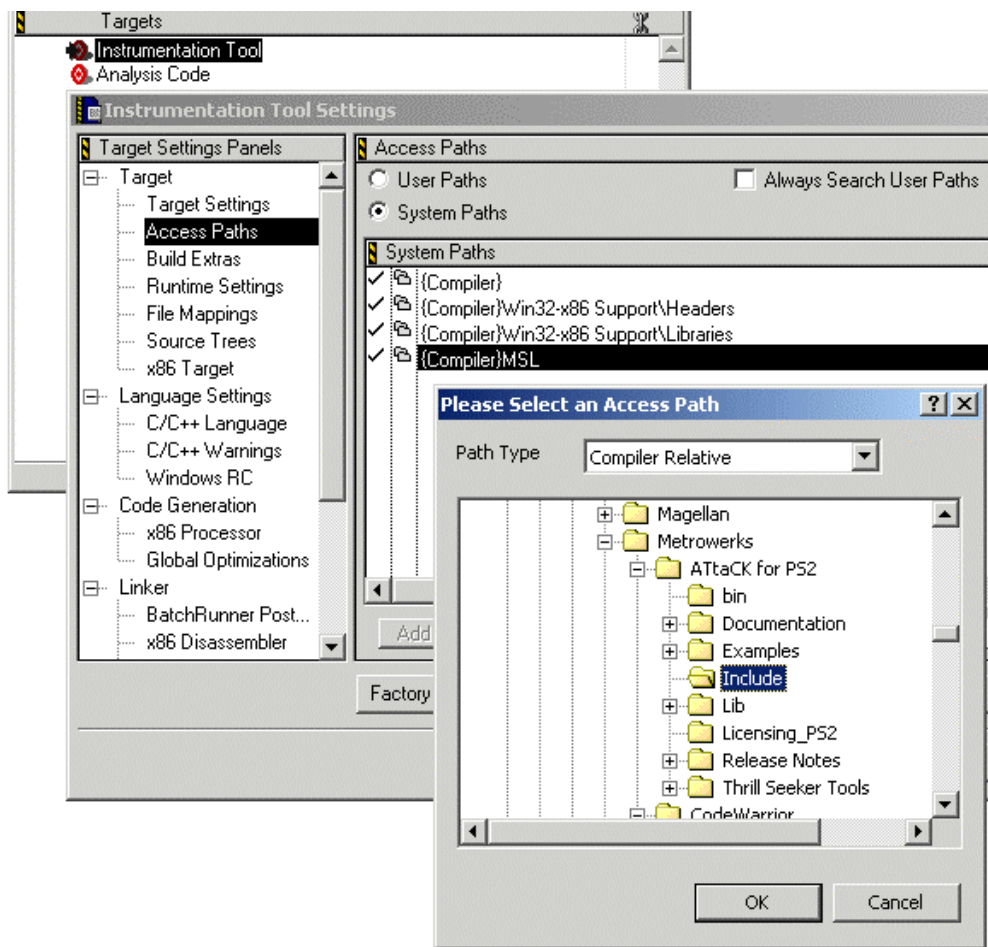
## Setting the Instrumentation-Tool Target

Double-click on the "Instrumentation Tool" target again. The target settings are grouped into "panels" of related options. You pick which panel to view and modify by clicking in the list in the left-hand pane. After modifying a panel, it's a good idea to save your changes by hitting the **Apply** button. (Hitting **OK** closes the dialog, so don't do that until we're completely done.)

The "Target Settings" panel only needs one modification: Click on the **Choose...** button to change the output directory from the project folder to the **Bin** subfolder. When you hit **OK**, the directory will now appear as `{Project}Bin`.

Next, select the "Access Paths" panel. This has two "sub-panels," one for user paths and one for system paths—the latter are the ones referenced by `#include <filename.h>`. Make sure the **User Paths** radio button is selected and hit **Add...** Use the folder list to browse for your project's **Shared** folder and hit **OK**. Then hit **Add...** again to find and add the ATtaCK library folder, **Lib\ps2**. (If you have a default ATtaCK installation, this folder will be `C:\Program Files\Metrowerks\ATtaCK for PS2\Lib\ps2`.) Before you hit **OK**, though, change the droplist at the top of the dialog from **Project Relative** to **Compiler Relative**.

Now change the radio button to **System Paths**. Add three more compiler-relative paths: **Win32-x86 Support\Headers**, **Win32-x86 Support\Libraries** and **MSL**. Then add the ATtaCK include folder (`C:\Program Files\Metrowerks\ATtaCK for PS2\Include`), also compiler-relative. The figure below shows how the paths should look:



**Figure 01-03: Setting the "Instrumentation Tool" Target's Access Paths**

The "Build Extras" panel is fine. On "Runtime Settings," change the working directory to `Bin`. On "File Mappings," add a new file extension by entering `.lib` in the "Extension" edit field, changing the "Compiler" droplist to `Lib Import x86`, and hitting `Add`.

"File Mappings" and "Source Trees" are also fine. On "x86 Target," make sure the project type is `Application (EXE)`. On the "C/C++ Language" panel, make sure everything is off except `Enable bool Support`. Now skip all the way down to the "x86 Linker" panel. Change the "SubSystem" from `Windows GUI` to `Windows CUI`—this is a command-line tool, not a GUI tool.

That's all the settings that need to change for this target, so hit `OK` to close the "Target Settings" dialog.

---

## Using ATtaCK without CodeWarrior

The ATtaCK library uses the standard format, so you can build ATtaCK tools with any Windows C/C++ compiler. However, you'll have to deal with two separate IDEs, and you lose all the advantages of a dual Windows/PlayStation 2 installation—your instrumentation tool and analysis code will have to live in separate projects. You're really much better off using CodeWarrior for both halves of your ATtaCK tools.

No matter what, you'll have to use CodeWarrior for PlayStation 2 to build analysis code.

---

## Adding Instrumentation - Tool Files

From the menu, select `Project > Create Group...` and create a file group called `ATtaCK Libraries`. Select that new group and go to `Project > Add Files...` Browse for the file `atck.lib` (located in your `Lib\ps2` folder). Note that you'll have to change the "show files of type" droplist from source files to library files.

Using the same process, create a group called `MSL ANSI Libraries`. This folder will hold the standard C libraries, so that we've got access to functions like `printf()`. Add the library files `ansix86d.lib` (found in the `MSL\MSL_C\MSL_Win32\Lib\x86` subdirectory of your CodeWarrior installation) and `mwcrtd.lib` (found in `Win32-x86 Support\Libraries\Runtime`).

Now create a `Win32 SDK Libraries` group, which will—you guessed it!—hold the Win32 SDK libraries. (Even if you don't think your application uses any Win32 functions, it actually does—the standard C libraries use them "behind the scenes," so you have to link to the Win32 libraries no matter what). To this group add `gdi32.lib`, `kernel32.lib` and `user32.lib`, all found in the `Win32-x86 Support\Libraries\Win32 SDK` subdirectory.

Finally, create a group called `Instrumentation Tool`, to hold the source files for the instrumentation tool. With that group highlighted, select `File > New...` from the menu, click on the "File" tab and select "Text File" from the list. Give the new file the name `insttool.c`, and click on `Set...` to place it in the `Inst` folder. Check the "Add to project" box, and make sure that the "Instrumentation Tool" target is checked but that the "Analysis Code" target is not. Hit `OK`.

We're basically done with this target, but let's do one more thing for polish. The `insttool.c` file should now be open. This file will get copied into every new project we create with this stationery, so we might as well save ourselves a little bit of work. Add the following code to the top of the file:

```
#include <atck.h>
#include <atckps2.h>
```

Close and save `insttool.c`.

## Setting the Analysis-Code Target

The steps to get the analysis-code target set properly are basically the same as for the instrumentation-tool target. Start by double-clicking the "Analysis Code" target.

Once again, on the "Target Settings" panel, change the output directory from the project folder to the `Bin` subfolder. Next, select the "Access Paths" panel, and add the same two folders to the user paths as were added for the instrumentation tool: `{Project}Shared` and `{Compiler}...\ATtaCK for PS2\Lib\ps2`.

Now change the radio button to `System Paths`. Here you only need to add two compiler-relative paths: the ATtaCK include folder (`{Compiler}ATtaCK for PS2\Include`) and the PS2 Support subdirectory of your CodeWarrior installation (`{Compiler}PS2 Support`).

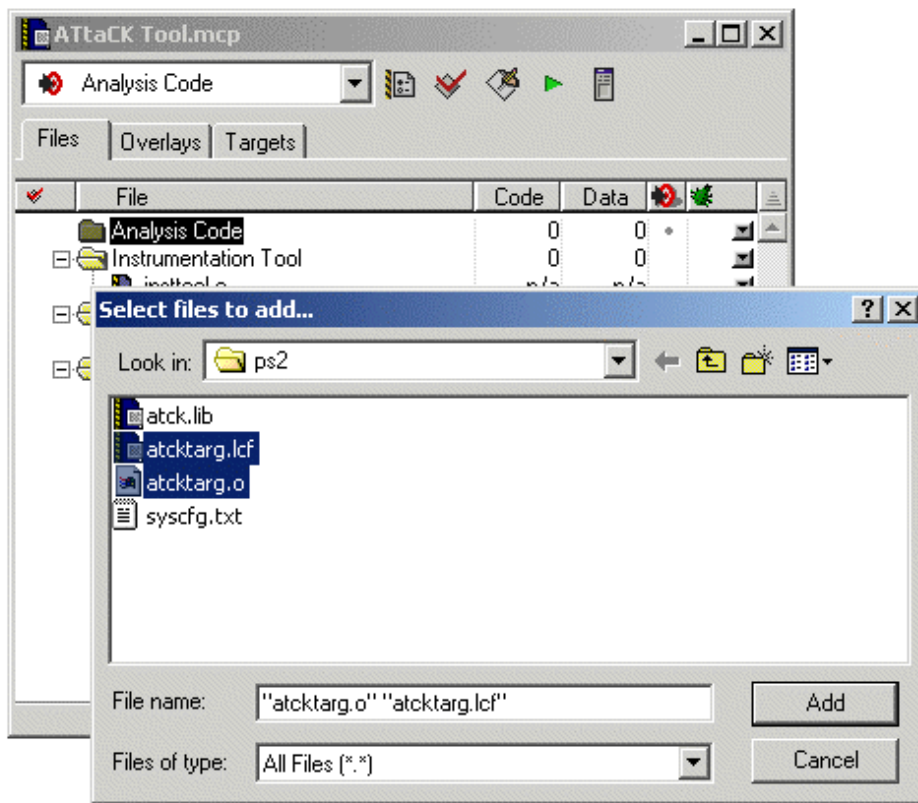
The "Build Extras" and "Runtime Settings" panels are fine, but "File Mappings" isn't. Add a new file extension by entering `.o` in the "Extension" edit field, changing the "Compiler" droplist to `Lib Import MIPS`, and hitting `Add`.

On the "MIPS Bare Target" panel, change the "Project Type" to `Application`, the "File Name" to `analcode.elf`, the "Byte Ordering" to `Little Endian`, and the "Small Data" threshold to `0`.

On the "C/C++ Language" panel, make sure everything is off except `Enable bool Support`. Now skip all the way down to the "MIPS Bare Linker" panel. Make sure "Generate ELF Symbol Table" is checked, and change "Entry Point" from `__start` to `atcktarget_start`. That's the last of the target settings, so hit `OK`.

## Adding Analysis-Code Files

In the "Files" tab, select the "ATtaCK Libraries" group and go to `Project > Add Files...` Browse for the `Lib\ps2` folder and add the files `atcktarget.lcf` and `atcktarget.o` to the "Analysis Code" target. (Again, remember that you will have to change the "show files of type" droplist to "all files" in order to see these files.)



**Figure 01-04: Adding Files to the Analysis-Code Project**

As we did with the instrumentation tool, we'll set up a source file to be created along with the project, ready to add code. Create a group called `Analysis Code`. With that group highlighted, select `File > New...` from the menu, click on the "File" tab and select "Text File" from the list. Give the new file the name `analcode.c`, and click on `Set...` to place it in the `Anal` folder. Check the "Add to project" box, and make sure that the "Analysis Code" target is checked but that the "Instrumentation Tool" target is not. Hit OK.

The new file will open for editing, so give it these contents:

```
#include <atcktarg.h>
#include <atcktargps2.h>
#pragma force_active on
```

You don't have to include any other headers for libraries—indeed you *can't* include them, because analysis code cannot access the Sony libraries. Likewise, you can't have a `main()` function in this file, because that's provided by ATtaCK. (You're free to *call* a function `main()` if you want, but it won't be the entry point for your analysis code.)

The `#pragma` directive is there to keep the linker from "optimizing away" all our code: Since it's the target application, not the analysis code, that calls the routines in this file, the linker doesn't think any of them are used, and will save space by throwing them away!

Close and save `analcode.c`, then close and save the entire project.

## Finishing the Stationery

To turn this project file into stationery, we have to clean up the project folder and move it to the CodeWarrior stationery folder. Use your favorite file shell to open the project folder. In addition to the folders you created, it should contain a folder CodeWarrior created called `ATtaCK_Tool_Data`. Delete that folder and all its contents, but leave the other folders alone.

Now move the `ATtaCK_Tool` project folder in its entirety into your CodeWarrior stationery folder. This folder is just off the main directory containing CodeWarrior, and will be named something like `C:\Program Files\Metrowerks\CodeWarrior\Stationery`.

And that's it! The next time you select `File > New...` in the CodeWarrior IDE, `ATtaCK_Tool` will be available as stationery for the new project. But don't take my word for it—go to the next section and we'll try it out!

## ***BareBones: The World's Least-Useful ATtaCK Tool***

We're going to jump right into some code, mostly to test your installation. Don't be intimidated! I'm not going to take the time right now to explain what each line does, except to say that every ATtaCK library routine begins with `atck_`, and every ATtaCK-supplied `typedef` begins with `atck_` and ends with `_t`. Just walk with me on how each of the steps discussed earlier is represented in the code below.

This tool is very simple, and its behavior is easy to describe. It iterates through every procedure in the target application, adding an instrumentation call to each one. The instrumentation call simply increments a counter. Thus, the program counts the number of function calls your target application makes.

Like I said, this is the least-useful ATtaCK tool in the world. But it'll establish whether you've got everything installed properly, and also give you an initial sample app so that you can see the basics of an ATtaCK tool.

## Creating BareBones

Let's make a project to make sure everything works. Select `File > New...`, highlighting the `ATtaCK_Tool` stationery, and name the new project `barebones`.

Open `analcode.c`. Replace its contents with that of `barebones-anal.c` in the "supplemental material" folder. Then open `insttool.c` and replace *its* contents with that of `barebones-inst.c`.

Now open the "Target Settings" dialog for the instrumentation-tool target, go to the "x86 Target" panel and change the name from `noname.exe` to something a little more appropriate, such as `barebones.exe`.

Save all that, and we're ready to see whether everything works.

## Compile-Time Troubleshooting

Select the `barebones.mcp` project window and hit F7. It should build one of the two targets, most likely the instrumentation tool. Once that target compiles properly, select the other target from the droplist at the top of the project window and build it. If errors occur during either of these builds, a window will open up. Don't worry about warnings—you'll get at least two warnings in your analysis-code project, but they don't matter. Here are some errors you may see:

### Undefined Symbols/Header File Not Found

Your access paths are wrong—check `Access Paths` under `Target Settings` for each project. If you're having trouble with relative paths, just cop out and use absolute paths.

### License Error

Your CodeWarrior for PlayStation 2 installation must be licensed to compile the analysis code. If you receive a license error, refer to your CodeWarrior documentation for more details.

### Anything Else

Check for typos in your code, although if you just copied what we gave you, it should be fine—we've checked! Also check to make sure that none of the standard PlayStation 2 libraries are included in your analysis-code project—the only files you should be linking to are ones you write yourself, and ATtaCK files (which all begin with "atck," naturally enough).

If you still have problems, you can reach Metrowerks tech support at [ps2\\_support@metrowerks.com](mailto:ps2_support@metrowerks.com), or call (800) 377-5416 in the U.S. - for customers outside the U.S. call +1(512) 997-4700.

## Instrumentation Troubleshooting

Now that you've successfully built the program, let's try it out! Open a DOS window and change to the project's `Bin` subdirectory, where you'll find `barebones.exe`. First, run it without an application at all, to make sure your environment is properly set.



Next, pick a PlayStation 2 application and type `barebones filename`—be sure to include the full path to your application! This should simply report a count of the number of procedures in the application. Note that this count will be higher than the number of routines in *your* code—even a simple "Hello, World!" program will have about 300 functions in it, thanks to the C runtime code, startup functions and other library routines.

Here are some errors you might encounter:

### **'barebones' is Not Recognized**

The file `barebones.exe` can't be found. Either you're in the wrong directory—you need to be in the `Instrumentation Tool` subdirectory of your `Barebones` project folder—or you compiled the analysis-code target rather than the instrumentation-tool target.

### **Unable to Locate DLL**

This error means your `DOS PATH` variable either hasn't been set or is pointing to the wrong directory. Type just `PATH` at the command prompt and make sure the `bin\ps2` subdirectory of your `ATtACK` folder is included properly.

### **Unable to Open Analcode.elf**

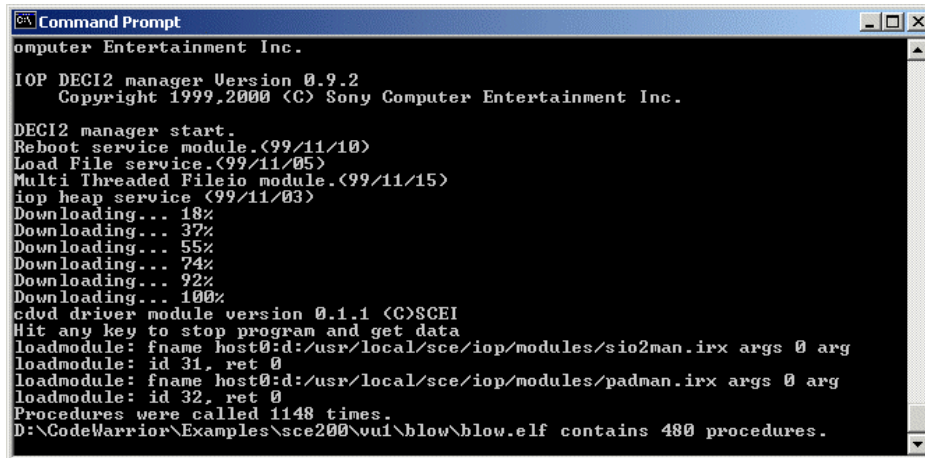
You probably compiled just the instrumentation-tool target without also building the analysis-code target, or you're running `barebones.exe` from somewhere other than the `Bin` directory.

### **License Error**

ATtACK will only function when licensed. This information should have come with you're your ATtACK installation package; if not, contact Metrowerks License Department at [license@metrowerks.com](mailto:license@metrowerks.com).

## **Run-Time Troubleshooting**

Now, using the same application, type `barebones -r filename`. This will give the same function count as before, but will also download the application to your T10000 and run it. Let the application run on the target for a while, and then hit any key to end the program. Your final output should look something like this:



```
Command Prompt
Sony Computer Entertainment Inc.
IOP DECI2 manager Version 0.9.2
Copyright 1999,2000 (C) Sony Computer Entertainment Inc.

DECI2 manager start.
Reboot service module.<99/11/10>
Load File service.<99/11/05>
Multi Threaded Fileio module.<99/11/15>
iop heap service <99/11/03>
Downloading... 18%
Downloading... 37%
Downloading... 55%
Downloading... 74%
Downloading... 92%
Downloading... 100%
cdvd driver module version 0.1.1 (C)>SCEI
Hit any key to stop program and get data
loadmodule: fname host0:d:/usr/local/sce/iop/modules/sio2man.irx args 0 arg
loadmodule: id 31, ret 0
loadmodule: fname host0:d:/usr/local/sce/iop/modules/padman.irx args 0 arg
loadmodule: id 32, ret 0
Procedures were called 1148 times.
D:\CodeWarrior\Examples\sce200\vu1\blow\blow.elf contains 480 procedures.
```

*Figure 01-05: Sample Output of Barebones.exe*

## Hangs

If the program seems to hang, it cannot find the T10000. Refer to `CWComUtil` and its documentation for debugging.

## What's Next

That's it for Lesson 01. In the next lesson, we'll start covering the ATtACK API, and you'll experiment more with the capabilities of this powerful framework.

---

### Debugging Analysis Code

The analysis code won't be part of the symbolic debugging info so normal debugging isn't possible. The code can't communicate with the target application or send messages to the host, so "printf debugging" is out. All you can do is step through the disassembly.

Moral: Keep your analysis code *simple*. Write as little code as possible, and make sure the code you do write is so straightforward that it can't conceal any bugs.

---

## Lesson 01 Quiz

This quiz will be a breeze, because this lesson mostly covered introductory material. You can expect there to be more questions in the next lesson!

1. The lessons are the most important part of this course. What's the second most important part of the course?
  - A The documentation.
  - B The example programs.
  - C Instructor interaction.
  - D A PlayStation 2 development system.
2. True or false: Regression testing is the most common method of detecting code errors.
  - A True
  - B False
3. What's the difference between using ATtaCK and simply writing analysis code directly into your program, for instance by using asserts?
  - A ATtaCK doesn't modify your program
  - B ATtaCK works with the binary image rather than the source code.
  - C ATtaCK doesn't require any work on your part.
  - D There is no difference.
4. Which team member should develop ATtaCK tools, and why?
  - A A senior programmer, because ATtaCK tools are very complicated.
  - B A junior programmer, because ATtaCK tools are very easy.
  - C A senior programmer, because analysis and optimization are critical.
5. True or false: ATtaCK analysis code is written and compiled just like any other PlayStation 2 program.
  - A True
  - B False

## Lesson 02: Examining an Application

---

In our second lesson, you'll learn how to use ATtaCK to break your application down into its basic elements. We'll then look at our first real ATtaCK tool.

### *Anatomy of an Application*

#### Lesson Objectives

If you'll remember our discussion of the ATtaCK process in the last lesson, you'll see that I promised we'd cover five steps in this lesson: initialization, reading system options, opening target applications, stepping through code using iterators, and closing ATtaCK.

I know, I know, five out of eleven sounds like a lot for one lesson. Really, though, we're going to spend most of our time walking through code. The rest, from initialization to shutdown, are just boilerplate housekeeping tasks that will take you about ten minutes to learn. You can't have an ATtaCK tool without that stuff, but studying it isn't very edifying, so we'll cover it as briefly as possible.

Code navigation, on the other hand, is the heart and soul of ATtaCK. At a minimum, your tools will need to step through the target program just to examine it, disassemble it and otherwise reveal its inner workings to you. "Real" tools, the kind your boss expects to justify ATtaCK's price tag, do more: They actually modify your application to insert analysis code.

That doesn't just happen by itself—you have to tell ATtaCK what code goes where. The "what code" part comes in Lesson 03. First, you need to become an expert on "where." And the first step toward that is to learn how ATtaCK models your application.

### **Anatomy of an Application**

Any analysis tool must model the target application in some logical way, so that you can examine it and identify the code you want to monitor. The IDE, for instance, models an application as a collection of files and lines. When you want to insert a breakpoint into your code, you do so by marking a particular line of a particular file.

The IDE presents your application as files and lines ultimately because that's the way *you* see it. The IDE is a tool for getting information out of your head and into the computer. ATtaCK, on the other hand, isn't particularly concerned with how you see things (sorry!). As a tool for getting information out of the CPU, ATtaCK models your application the way the CPU sees it: as a series of **instructions**.

Now, a large application might have hundreds of thousands of instructions, which would get tedious to step through. Fortunately, there's a shortcut: As long as a sequence of instructions is only entered at the beginning and only exited at the end, you can just treat it as one mega-instruction—a **basic block**. Apart from the handful of instructions that take a variable number of cycles to execute, you can

assume that a single basic block will always consume the same amount of time and have the same potential side effects every time it runs.

Let's pause here for me to point out that ATtaCK can break down your application into basic blocks reliably only because it looks at every branch and call in the entire program. ATtaCK is omniscient and infallible. It's not possible for your program to ever jump into the middle of a basic block, because if any jump in the entire application targeted the middle of that block, ATtaCK would have split the block in two. This is just one small example of the very extensive processing that ATtaCK does behind the scenes. ATtaCK tools are easy to write; ATtaCK itself most assuredly was not!

---

## ATtaCK's Limitations

---

ATtaCK is designed to be language-agnostic, but it's really most comfortable with C or assembly language. C++ presents a few "gotchas" to watch out for.

First of all, whenever you use ATtaCK to read the name of a C++ function, you'll have to unmangle the name yourself.

Second, C++ virtual methods are called indirectly, using vtables—ATtaCK knows where indirect calls take place, but can't tell what methods are getting called.

---

## Procedures

Moving up the hierarchy, basic blocks can themselves be grouped together. Your application is almost certainly structured into functions or **procedures**. ATtaCK defines a procedure as a collection of basic blocks that, once entered, is only left via one or more known exit points. The procedure includes all the basic blocks that are branched to, but not those that are called. (Just to review the distinction: A *call* transfers execution with the expectation that the program will return to the instruction following the call; a *branch* transfers execution with no expectation of coming back.)

Their **entry points** define procedures. An entry point is ATtaCK's term for the target of a procedure call. Every procedure must have at least one entry point; if it didn't, it wouldn't be a procedure. ATtaCK finds procedures by reading the symbol table and looking at the target of every call in the program. This means that a procedure in the symbol table will show up in ATtaCK even if it's never called. Conversely, a procedure with no entry in the symbol table—for instance, a relative call within an assembly module—will also show up in ATtaCK, although it won't have a symbolic name, of course.

If your application is written in C/C++, each procedure has one and only one entry point, at the opening curly brace. The situation in assembly is more complicated: A procedure could have multiple entry points, and might not even have an entry point at the beginning. How? Well, it depends on your definition of the word "beginning." The execution of a procedure must always begin at an entry point. However, when ATtaCK gathers together all the basic blocks of that procedure, it sorts them by ascending address. If the procedure branches to a

block with a lower address than the entry point, that block will be the beginning of the sorted list.

Assembly procedures present other opportunities for confusion. Let's say your program has two functions that perform the same basic clean-up tasks before returning. To save space, you end each of those functions not with a return but with a branch to the common final code, which performs the clean-up and then executes the actual return. From your point of view, these are two separate procedures. ATtaCK, on the other hand, sees them as one single procedure with two entry points. The rule is that each basic block belongs to one and only one procedure; any time two or more procedures can reach a basic block, ATtaCK folds those together into one.

## The Rest of the Story

There are just a few ATtaCK terms left before the model is complete.

A **call site** is just a function call. Most of these will be to procedures within your program, in which case there will be an ATtaCK-identified entry point at the target of the call. When you call OS code that lives outside your application, there won't be an entry point, although you will be able to query for the target address. When you call a virtual function, ATtaCK doesn't even know the target address.

An **image** comprises all the basic blocks that share a single address space. For example, in a Windows program, the application itself would be one image, while each .DLL it loaded would be another. On the PlayStation®2, each code overlay would be a separate image. However, the current version of ATtaCK doesn't support code overlays, so you'll only see one image per program.

Finally, the **program** is your application itself. As mentioned before, on the PlayStation 2 there's a one-to-one relationship between programs and images, but ATtaCK still treats the two separately.

## Top-Down Review

It's easier to define each part of the ATtaCK model by starting at the bottom and working up. On the other hand, it's easier to get the big picture by starting at the top and working down, so let's review that way:

Your application is a **program**.

Each program is made up of one or more **images**; on the PlayStation 2, there can be only one.

Each image is made up of **procedures**, which are defined by the presence of one or more **entry points**. Entry points are usually the targets of **call sites**.

Each procedure is made up of **basic blocks**.

Each basic block is made up of **instructions**.

This top-down structure is exactly how you'll use ATtaCK to step through your application: First you get a handle to the program, then you use that to get a handle to an image, then you use *that* to get a handle to a procedure, and so forth.

## ATtaCK Types: Objects

### ATtaCK Objects

ATtaCK uses objects internally to maintain its state information. Every ATtaCK function either acts upon one or more of these objects, or creates and returns a new object; most do both!

Pointers to opaque types represent these objects. For example, a session object is represented by a pointer to type `atck_sesn_t`, not by a variable or structure of type `atck_sesn_t`. Your program should *never* try to instantiate the underlying type directly, nor should it ever de-reference, `delete` or `free()` one of these pointers. Always use the specific ATtaCK functions to manipulate objects. To emphasize this, we'll always refer to these pointers as **handles**.

You might want to create a small header file that defines an opaque type for each handle. For example:

```
typedef atck_sesn_t* HSESSION;
```

will allow you to subsequently declare variables such as

```
HSESSION hSession;
```

Depending on your personal coding style, this may be clearer to you. It certainly is to me, but then my mind has been warped by a decade's worth of Windows programming. Remember that the purpose of ATtaCK is to enable you to create new, custom analysis tools quickly. If wrapping the ATtaCK handles in your own types, obeying your own naming convention, helps you do that then by all means, go right ahead!

### "Methods" and "Attributes"

Conceptually, ATtaCK objects have methods, functions that operate on their data. They also have attributes, data they expose to your program. Since the ATtaCK API is written in straight C, however, these methods and attributes aren't implemented as members of the objects themselves, but are instead separate, global functions.

Almost every ATtaCK function takes, as its first argument, a handle to an ATtaCK object. For example, the function to open a new application, `atck_open()`, takes as its first argument a handle to a session object. `atck_open()` is thus "a method of the session object," even though as far as the compiler's concerned it's not a member of anything.

Similarly, object attributes are not exposed as C-style structure member variables. Instead, each attribute has an access method. For example, the number of images contained within a program is an "attribute" of the program object. This value is returned from the access function `atck_nimg()`, which takes as its first (and only) argument handle to a program object.

At the risk of some short-term confusion, we'll continue to call these functions "methods" and "attributes," even though from the compiler's perspective they're

nothing of the kind. As you learn more about the ATtaCK API, you'll see how this concept makes it much easier to remember each function's name and arguments.

## Code Objects

Code objects represent the "anatomical elements" we discussed in the previous section: program, image, procedure, entry point, call site, basic block and instruction. These objects have attributes, which are used to examine your application. For instance, the `atck_ent_t` object contains the address, symbolic name, file name and line number of a given entry point, and the handle of the procedure object that contains the entry point.

These objects are created invisibly as needed as you step through the code. You do not need to release them—they are owned by their parent object, and freed when it is freed. The program is the parent of each image, while the images are the parents of every other object.

Handles are unique and consistent. You can never have two different handles for the same object at the same time. For example, if you use two iterators to step through the basic blocks in a procedure, both iterators will return the same sequence of handles. This means that you can simply test two handles for equality to see whether they reference the same underlying structure in the application. Handles are *not* persistent, however. If you run the tool twice in a row, you will get different handles for each run.

Code objects will be covered in detail later in this lesson.

| Code Object | Abbreviation      | Object Type              |
|-------------|-------------------|--------------------------|
| Program     | None (see below)  | <code>atck_prog_t</code> |
| Image       | <code>img</code>  | <code>atck_img_t</code>  |
| Procedure   | <code>proc</code> | <code>atck_proc_t</code> |
| Entry Point | <code>ent</code>  | <code>atck_ent_t</code>  |
| Call Site   | <code>call</code> | <code>atck_call_t</code> |
| Basic Block | <code>bb</code>   | <code>atck_bb_t</code>   |
| Instruction | <code>inst</code> | <code>atck_inst_t</code> |

**Table 02-01: Code Objects**

The abbreviation listed for each object is used throughout ATtaCK, such as in the object's type name. The abbreviation is also used to identify that object's methods. For example, every method of the call site object is of the form `atck_call_methodname()`. The program object is a special case here: It is the "default" for methods, so methods of the program object are of the form `atck_methodname()`, not `atck_prog_methodname()`.



## State Objects

These objects maintain the state information for ATtaCK. You must create and destroy them yourself, using the ATtaCK functions provided.

| Type   | Definition  | Covered In |
|--|---|------------|
| atk_sesn_t   | Session, the top-level ATtaCK object  | Lesson 02  |
| atk_run_t  | A program downloaded to the target system   | Lesson 04  |
| atk_reg_t  | A set of registers referenced by an instruction   | Lesson 05  |
| atk_imgit_t<br>atk_procit_t<br>atk_entit_t<br>atk_callit_t<br>atk_bbit_t<br>atk_instit_t | Iterators to step through images, procedures, entities, call sites, basic blocks and instructions, respectively | Lesson 02  |

*Table 02-02: State Objects*

## Other Objects

These objects are, like code objects, maintained by ATtaCK. You don't have to destroy them, although in some cases you have the option of doing so.

| Type        | Definition   | Covered In |
|-------------|--|------------|
| atk_cfg_t   | A set of system configuration options                            | Lesson 02  |
| atk_dev_t   | A device connection, through which you control the target system | Lesson 04  |
| atk_iprog_t | An instrumented program ready to download and execute            | Lesson 04  |
| atk_tevt_t  | An event raised on the target machine and sent to the host       | Lesson 04  |

*Table 02-03: Other Objects*

## What About C++?

I'm sure the C++ programmers out there are already jumping up and down, asking "Why aren't these *real* objects?!" The answer is "No reason why not—be our guest."

Seriously, there would be a lot of advantages to wrapping the ATtaCK API in a set of C++ objects. For one thing, you'd get much stronger type checking, which

would help prevent mistakes. The API would also be much cleaner: `entry.GetAddress()` is easier to understand than `atck_ent_addr(pent)`. The API's pretty simple and consistent now, though, so those aren't killer advantages.

What *would* be killer is the convenience of constructors and destructors. Not only would these simplify the API for the various cleanup functions, they'd also handle the "boilerplate" code invisibly. For instance, the default `ATCKSession` constructor could automatically load the default system configuration file, a bit of code that basically never changes. A quickie set of wrapper classes I slapped together reduced the size of Lesson 01's `BareBones.c` by 20%, and the more complex your iteration tasks the more savings you'd see.

Wrapping ATtaCK in C++ isn't all peaches and cream. Not everyone is comfortable with C++, so you might be reducing the pool of programmers able to write and maintain your ATtaCK tools. More to the point, creating a proper, well-conceived set of C++ wrappers would take time away from actually *using* the tools. Quickie classes like the ones I slapped together will probably do you more harm than good; C++ is not very forgiving of programmers who don't plan ahead.

So what's the bottom line? If your shop already uses C++, and you plan on using ATtaCK for more than one project, then you should seriously think about creating C++ wrappers. It'll probably be a draw on the first project, and then pay for itself every project after that. But ATtaCK really is easy to use even in its straight-C incarnation; so don't feel like you're missing out if you stick to the plain vanilla API.

Now, if you want something *really* bleeding-edge, consider this: By wrapping the ATtaCK functions in a set of COM objects, you could write your analysis tools in scripting languages such as Perl, Python, JavaScript—anything for which a Windows Scripting Host interpreter exists. I'm not sure you really gain anything by doing that, but it'd be cool!

## ***ATtaCK Types: Data Types***

### **Fundamental Types**

ATtaCK is designed for cross-platform use. Obviously there are two platforms right off the bat: the PC, where the instrumentation tool runs, and the PlayStation 2, where the target application runs. Beyond that, future versions of ATtaCK may run on different host platforms, such as the Mac, or may analyze different target platforms, such as next-generation consoles. By and large, any ATtaCK tools you create today will run with little or no modification on future platforms.

Toward that end, ATtaCK provides the usual suspects: fundamental types to represent `ints`, `bools` and `floats` of various sizes. There are also constants defined to show the maximum ranges for these types. These types and constants are summarized in the table below.

| Type                        | Definition                    | Range                          |
|-----------------------------|-------------------------------|--------------------------------|
| <code>atck_int16_t</code>   | 16-bit signed integer         | ATCK_INT16MIN to ATCK_INT16MAX |
| <code>atck_int32_t</code>   | 32-bit signed integer         | ATCK_INT32MIN to ATCK_INT32MAX |
| <code>atck_int64_t</code>   | 64-bit signed integer         | ATCK_INT64MIN to ATCK_INT64MAX |
| <code>atck_uint64_t</code>  | 16-bit unsigned integer       | 0 to ATCK_UINT16MAX            |
| <code>atck_uint32_t</code>  | 32-bit unsigned integer       | 0 to ATCK_UINT32MAX            |
| <code>atck_uint64_t</code>  | 64-bit unsigned integer       | 0 to ATCK_UINT64MAX            |
| <code>atck_bool_t</code>    | A boolean of unspecified size | Either ATCK_TRUE or ATCK_FALSE |
| <code>atck_float32_t</code> | A four-byte float             | N/A                            |
| <code>atck_float64_t</code> | An eight-byte float           | N/A                            |

*Table 02-04: Fundamental Types*

## Target Addresses

On the PlayStation 2, addresses are 32 bits, the same size as on the PC. However, that might change for other targets and hosts in the future. To handle this properly, always use the `atck_addr_t` type to hold target addresses; its value can range from 0 to `ATCK_ADDRMAX`. Although `atck_addr_t` is essentially just an `atck_uint32_t`, you should always use the special address type. Never use an integer type, and especially never use a host pointer type, to store target addresses; no good can come of it.

## Target Mutex Variables

In order to protect critical sections on the target, and to handle multithreaded target code in general, ATtaCK provides a mutual-exclusion lock system for analysis code. These mutex variables are always of the type `atcktarg_lock_t`, which should be treated as an opaque fundamental type. Mutex locks are covered in Lesson 04.

## Enumerated Types

### Flags

Many ATtaCK functions take flag-like arguments or have flag-like return values. ATtaCK stores these using enumerated types. One example of this is discussed in the notes below, the `atck_endian_t` used by `atck_byteorder()`. Since these

types really only have meaning in the context of their functions that's where we'll discuss them. For a consolidated reference to them, see page ATK-45 of the online documentation.

---

## Endianism

The PlayStation 2 and PC are both little-endian, but some target/host combinations may have different byte-ordering. It only matters for values that cross the host-target boundary. The `atck_addr_t` type, for instance, is always stored in *host* byte order, even when it holds an address on the *target*.

Your analysis code runs on the target, but its information gets uploaded to the host. Even then, ATtACK can automatically resolve any byte-order differences for you—see Lesson 04.

---

## Opcodes and Registers

You can query an instruction for its opcode and the registers that it uses. The `atck_op_t` type represents a "pseudo opcode" value for an instruction, which identifies the function performed by the instruction but doesn't necessarily correspond to an actual hardware opcode value for the target processor—although for the PlayStation 2 they basically do. Similarly, `atck_reg_t` identifies a target processor register.

These types will be discussed more when we look at instruction attributes later in this lesson. They're also described in the online documentation—see page ATK-232 for opcodes and page ATK-50 for registers.

## Startup and Shutdown

Now that you've got the big picture, we'll talk about building an ATtACK tool. The very least you have to do is initialize ATtACK at the beginning of the program and close it down at the end, so we'll start with that. You might want to refer back to BareBones.c as we go.

## Error Handling

A function that returns a handle will return `NULL` to indicate an error. A function that returns a fundamental type will typically return 0 or `ATCK_FALSE` in case of an error. In either case, the framework will also write a diagnostic message to `stderr`. Since ATtACK tools are just simple command-line programs written for use by programmers, the most appropriate response to an error is usually just to exit.

## Startup Tasks

### Initialize an ATtACK Session

```
atck_sesn_t* atck_session(const char* ver, atck_flags_t
flags,...)
```

This function must be called before any other ATtaCK functions. It initializes the framework and returns the resulting session object. In the future, `atk_session()` might take multiple parameters, but right now it accepts only two: *ver*, a text string specifying the version of ATtaCK the tool expects, and *flags*, a bit-flags integer that controls the behavior of the session. These arguments have only one legal value each: *ver* must be the string "1.0", and *flags* must be the constant `ATCK_FLAGS_NONE`.

Save off the returned handle, because as you'll see, just about every other function in ATtaCK requires it.

## Open Application

```
atk_prog_t* atk_open(atk_sesn_t* session, const char*
appfile,
                                const char* analysisfile, atk_cfg_t*
config,
                                const char* outfile)
```

This function opens *appfile*, the target application. The returned program handle can be used to iterate through the application's code.

The parameter *analysisfile* is the filename of the compiled target analysis code that will be used by the instrumented application. This topic is covered in Lesson 03. If you will not be adding any instrumentation code to the application, you may pass in `NULL` instead.

The parameter *outfile* is the filename template to use when writing out the newly-instrumented images. Since an application might have more than one image, the template allows you to specify a naming scheme by which new image filenames can be generated as necessary.

The format for this template is covered in the documentation on page ATK-69 of the documentation that came with your ATtaCK installation. However, here's a boilerplate definition that will almost always work for you:

```
"<dir><base>_toolname<suf>"
```

This template, which is used by all the example programs, places the new executable in the same directory as the original one, adding "*\_toolname*" to the end of the base name but keeping the file extension the same. Replace *toolname* with your tool's name, such as `proccount`.

If you intend to download and execute the instrumented program immediately, and are content to throw the program away once the profiling session is done, you may specify `NULL` for the output filename. ATtaCK will then use temporary files to hold the instrumented program.

If you will not be adding any instrumentation code to the application, you may pass in `NULL` instead. You may also pass `NULL` if you intend to immediately download and run the application after instrumenting it. In this case, ATtaCK uses temporary files to hold the instrumented program, deleting them after your tool exits.

Finally, the parameter *config* would be used to specify a set of configuration options. However, the PlayStation 2 version of ATtaCK doesn't recognize any such options for use with `atck_open()`—there's only one way to open a program!—and so you should just pass `NULL`.

## Shutdown Tasks

### Write Out or Close the Application

```
void atck_img_release(atck_img_t* image)
```

This function closes the specified image, releasing all the procedures, entry points, call sites, basic blocks and instructions it contained.

If you add instrumentation code to an image, close it with `atck_img_write()` instead, discussed in Lesson 03.

### Close ATtaCK

```
void atck_endsession(atck_sesn_t* session)
```

This function closes the active session, shutting down ATtaCK. After you call this, you may still work with and display the data your tool has gathered, but no ATtaCK functions will be available.

As discussed earlier in this lesson, any state objects you create—in particular, iterators—must be freed individually. You must also close image objects individually. You don't have to worry about any other objects, though, because they belong to and will be freed by the session.

## Navigating with Iterators

Okay, you've successfully initialized ATtaCK and your application is open. Now what? Well, mostly you iterate.

### Step Through Code Using Iterators

The textbook definition of an iterator is "an object associated with a list that is capable of traversing the list, accessing the elements one at a time." That pretty much hits the nail on the head, so let's look at the definition one piece at a time.

- *"an object associated with a list..."* When ATtaCK loads the target application, it runs through it and builds lists of its images, procedures, entry points, call sites, basic blocks and instructions. There is an iterator type associated with each one of these lists.
- *"... that is capable of traversing the list..."* ATtaCK iterator objects have methods that move to the head or tail of the list, and that step forward and backward.
- *"... accessing the elements one at a time."* The traversal methods each return a handle to one code element from the list. For instance, every time you tell a

procedure iterator to "move next," it returns a handle to the next procedure in the list.

## Iterator Types

There are iterator types for each of the six lists, as shown in the table below. Remember that, as with all ATtACK objects, you only deal with *handles* to these objects, so to declare an iterator variable you'd use:

```
atck_procit_t* procedures;
```

Iterator types all follow a consistent naming convention, `atck_objit_t`, where *obj* is the abbreviation for the object type returned by the iterator. The image iterator, for instance, is `atck_imgit_t`.

## Iterator Methods

The iterator objects have methods but not attributes. The function names all follow a consistent pattern: `atck_objit_action()`. For example, the "move next" method for a procedure ("proc") iterator is called `atck_procit_next()`, while the same method for a basic-block ("bb") iterator is `atck_bbit_next()`.

As methods, these functions always take a handle to the iterator object itself as their first and only argument.

```
atck_obj_t* atck_objit_first(atck_objit_t* self)
atck_obj_t* atck_objit_last(atck_objit_t* self)
```

The `_first()` method returns the first element in the list, while `_last()` returns the last element in the list. If and only if the list is empty, both functions will return `NULL`.

```
atck_obj_t* atck_objit_next(atck_objit_t* self)
atck_obj_t* atck_objit_prev(atck_objit_t* self)
```

The `_next()` method returns the next element after the element most recently returned by the iterator, or the first element if the iterator was just created. If the most recent element returned was the last in the list, or if the list is empty, `_next()` returns `NULL`.

The `_prev()` method returns the previous element before the element most recently returned by the iterator. If the most recent element was the first in the list, or if the iterator was just created, or if the list is empty, `_prev()` returns `NULL`.

You might be wondering how a list could be empty—after all, the target application is pretty much guaranteed to have one of every code element, right? Well, that's true, but ATtACK may not be able to read into every procedure in your application. If, for example, a badly-formed call instruction wound up pointing to your application's global variable space rather than to actual code, the resulting "procedure" would probably contain illegal instructions and general nonsense.

ATtACK's a pretty no-nonsense framework, so faced with a situation like that it just marks the procedure as "unreadable" (`ATCK_ATTR_NOREAD`) and moves on.

The procedure object itself is still available, but many of its attributes will return zero or NULL. If you request any iterator from that procedure, or a basic-block iterator from an entry point contained in that procedure, the iterator will be empty.

## Creating Iterators

Each list is an attribute of its containing object. For example, the list of images in a program is an attribute of the program object. You'd thus expect that the functions to create new iterators are methods of the containing objects—and you'd be right!

```
atck_imgit_t* atck_imgit_new(atck_prog_t* self)
atck_procit_t* atck_img_procit_new(atck_img_t* self)
atck_entit_t* atck_proc_entit_new(atck_proc_t* self)
atck_callit_t* atck_proc_callit_new(atck_proc_t* self)
```

By now, these functions should be self-explanatory. The only question is sorting:

- Since PlayStation 2 applications only have one image, the list will only have that one entry, so you don't have to worry about how images are sorted.
- Procedure, entry-point and call-site lists are sorted by starting address in ascending order. Note that for call sites, the sort is done on the address of the site, not the address of the target.

```
atck_bbit_t* atck_proc_bbit_new(atck_proc_t* self)
atck_bbit_t* atck_img_bbit_new(atck_img_t* self)
atck_bbit_t* atck_ent_bbit_new(atck_ent_t* self)
```

There are actually three ways to iterate across basic blocks. You can iterate through all the blocks in a procedure, or all the blocks in an image. You can also iterate through a range of the blocks in a procedure, starting at a specific entry point and continuing to the end. (This allows you to trace execution from an entry point; without it, you'd have to step out to the procedure and walk its block list, as discussed in the next section.) In all three cases, the blocks are sorted by starting address.

```
atck_instit_t* atck_bb_instit_new(atck_bb_t* self,
                                atck_flags_t flags)
```

Because instruction lists can be quite large, ATtACK only creates them when necessary. This function will create and return an iterator for the list of instructions in the specified basic block. If the *flags* parameter is `ATCK_FLAGS_ITLIFE`, the instruction handles returned by the new iterator will only last as long as the iterator itself does; when the iterator is released, the instruction objects will also be released and their handles will no longer be valid. If *flags* is `ATCK_FLAGS_NONE`, then the instruction objects will have the same lifespan as all code elements, lasting until their containing image is released.



## Releasing Iterators

```
void atck_objit_free(atck_objit_t* self)
```

The "destructor" for an iterator object is the `_free()` method. You are always responsible for releasing any iterators you create. When you release an image, all the code objects it contains will be released, and their handles will become invalid. The iterators are *not* released, however. Good practice is to release iterators as soon as you're done with them, and in any case to release them before calling `atck_img_write()` or `atck_img_release()`.

## Finding Code Structures

Iteration is important, and just about every ATtACK programming task involves working with iterators. But to write any advanced tool, you'll need to do more than just iterate through the application from beginning to end. You might need to find the target of a call, or look up a specific named routine, or follow an execution path between two basic blocks.

Often, the information you need will be available as an attribute of some other object—a handle to the target entry point of a call site, for instance, is stored as part of the call site object.

Some tasks that require searching are nevertheless so common that ATtACK provides "lookup" methods to do the job for you. Finding a specific named routine is of course one such task, and is handled by the functions

```
atck_ent_byname() and atck_img_ent_byname().
```

All other search tasks must be performed by hand, using a technique called "iterate-and-query": Iterate across all the candidate objects, checking each one's attributes against some criteria until you find a match.

As you've guessed, "iterate and query" is just a more poetic term for "brute force."

## Link Attributes

Link attributes join one object to another. Like all attributes, they are retrieved using attribute access methods, which follow a simple naming convention:

`atck_obj_attribute()`. The method to get the target entry point for a call site, for example, is `atck_call_target()`.

## Parent Handles

These attributes store the handle of an object's parent.

```
atck_img_t* atck_proc_img(atck_proc_t* self)
atck_proc_t* atck_ent_proc(atck_ent_t* self)
atck_proc_t* atck_call_proc(atck_call_t* self)
atck_proc_t* atck_bb_proc(atck_bb_t* self)
atck_bb_t* atck_inst_bb(atck_inst_t* self)
```

Every code object has as one of its attributes a handle to its parent. Well, okay, not *every* code object contains a parent handle—there’s only one program object, so you’re expected to be able to remember the parent of image objects yourself. Below that, though, each procedure points to its containing image; each entry point, call site and basic block points to its containing procedure; and each instruction points to its containing basic block. These functions can never return `NULL`, since each of these objects will always have one and only one parent.

## Target Handles

These attributes store the handle of an object’s target.

```
atck_ent_t* atck_call_targent(atck_call_t* self)
```

This attribute stores a handle to the entry point in the target procedure called by this call site. Some call sites don’t target entry points, either because they call dynamic addresses (e.g, C++ virtual functions) or because they call addresses outside the target application. If ATtaCK can’t determine the target of a call site statically, this attribute will be `NULL`.

```
atck_bb_t* atck_inst_branchtarg(atck_inst_t* self)
```

If an instruction conditionally or unconditionally branches to another instruction within the same image, this attribute will hold a handle to the basic block targeted by the jump. As always, ATtaCK can’t see the future, so this attribute is `NULL` for dynamic jump instructions (such as the MIPS `JR`). Note that calls are a special kind of branch as far as ATtaCK is concerned, so this attribute is valid for instructions that call other procedures as well as for branches within one procedure.

This might seem like a useful method to walk through code: Iterate across a basic block, then call `atck_inst_branchtarg()` to get the next block when the instruction iterator returns `NULL`. But there’s a gotcha! Some chips—such as, coincidentally enough, the EE—have delayed branches, where the instruction immediately after a branch is executed while the chip performs the jump. On such architectures, the instruction following a conditional branch is still part of the first basic block. Thus, the last instruction of a basic block is *not* guaranteed to be either a call or a branch.

This attribute is not guaranteed to be valid even for branch instructions. A procedure marked `ATCK_ATTR_NOREAD` has no basic blocks. If the branch target lies in such a procedure, this attribute will be `NULL`.

## Alias Handles

These attributes store the handle of an object’s alias—that is, an object of a different type with which it has a one-to-one relationship. An instruction object that represents a procedure call, for instance, has a one-to-one relationship with the call site object that also represents that call.

```
atck_ent_t* atck_bb_entry(atck_bb_t* self)
```

If a basic block contains an entry point to a procedure, this attribute will hold a handle to that entry point; otherwise it will be `NULL`. Remember that code will never jump into the middle of a basic block, so if a block contains an entry point, that point will be the first instruction in the block. For the same reason, a block cannot contain more than one entry point.

As natural and handy as it might seem, there is no corresponding `atck_ent_bb()` method: Entry points are children of procedures, not basic blocks, even though each entry point is contained by one and only one basic block.

```
atck_call_t* atck_inst_callsite(atck_inst_t* self)
```

An instruction that conditionally or unconditionally calls another procedure *is* a call site, and will keep a handle to that call site object in this attribute. For all other instructions, this attribute will be `NULL`. As with basic blocks and entry points, this relationship is one-way: A call site does not point to the instruction that embodies it.

Even dynamic calls, where the target address cannot be determined by ATtaCK's static analysis, still have call sites, so this attribute will always contain a valid handle as long as the instruction is a call.

## Lookup Methods

These functions obey the naming convention `atck_obj_what_how()`. The method of the image object that looks up an entry point by its address, for instance, is called `atck_img_ent_byaddr()`. Remember that program is the default object, so methods of the program object are named `atck_what_how()`, not `atck_prog_what_how()`.

## Symbolic Lookups

ATtaCK gets its information two ways: examining the raw machine code directly, and reading the debugging information ("symbol table") from the executable file. You can run ATtaCK tools against "release builds" that don't contain debugging information, but you lose access to methods like these, which look up names in the symbol table.

```
atck_ent_t* atck_img_ent_byname(atck_img_t* self, const char*
name)

atck_ent_t* atck_ent_byname(atck_prog_t* self, const char*
name)
```

These methods examine the symbol table of the program or image respectively to find the entry point for the specified name. There might well be more than one entry point with the same symbolic name. In that case, only one entry point will be returned, in the following search order:

- The globally visible entry points are searched first, in the order that you'd see them if you iterated across every image, then across every procedure, then across every entry point. This means that the chosen entry point will

be the one with the lowest address contained in the procedure with the lowest starting address.

- If there are no globally visible entry points with the specified name, then *every* entry point is searched, using the exact same process.

If there is no entry point with the specified name in the program or image, this method returns `NULL`.

## Address Lookups

```
atck_ent_t* atck_img_ent_byaddr(atck_img_t* self, atck_addr_t
addr)

atck_inst_t* atck_img_inst_byaddr(atck_img_t* self,
                                atck_addr_t addr)
```

If *addr* corresponds to the start of an entry point or instruction respectively in the specified image, this method returns that object's handle; otherwise it returns `NULL`.

Procedures marked `ATCK_ATTR_NOREAD` cannot have any instruction objects. If *addr* falls within such a procedure, `atck_img_inst_byaddr()` will return `NULL`. Even unreadable procedures have entry points, however, so `atck_img_ent_byaddr()` will still work.

## Symbol Translation

```
atck_addr_t atck_symaddr(atck_prog_t* self, const char* name)

atck_addr_t atck_img_symaddr(atck_img_t* self, const char*
name)
```

Rather than finding a code object handle, these methods return the address of the specified symbol. The symbol might be a data variable or a procedure. If the symbol can't be found, these methods return zero.

## Iterate and Query

ATtaCK is built around iteration. In fact, ATtaCK is really built around *inward* iteration: It's easy to go from an object to the objects it contains, but much less easy to move from one object to one of its peers. For example, let's suppose all you have is an instruction handle. How do you get the next instruction?

One way would be to get the instruction's address with `atck_inst_addr()`, add its size from `atck_inst_rawsize()` to get the address of the next instruction, then use `atck_img_inst_byaddr()` to get the handle. The problem is that you're not really following the flow of execution, you're just reading the code blindly.

A better solution would be to use iterators. First, call `atck_inst_bb()` to get the instruction's basic block. Then use an instruction iterator to step through the block until you find a handle that matches the one you already have. (Remember: Two handles that point to the same object will always have the same value.)

Now when you call `atck_instit_next()`, you have the next instruction in the flow of execution, and you can continue iterating. To continue walking once

you've reached the end of this iterator, you use `atck_bb_proc()` to get the procedure, create a basic-block iterator from the procedure and use the same iterate-and-compare technique to find your current basic-block handle in the procedure's list.

As you can see, the process isn't hard to figure out, it's just a little tedious to program. One moral to this story is to remember your context as much as possible. Since going from an iterator to an object it contains is *much* easier than going from an object to the parent iterator that generated it, your tool's functions should pass iterators and high-level objects around in preference to low-level objects.

## Querying Code Attributes

All code objects have attributes. As we've seen, attributes are accessed using attribute access methods, which follow a simple naming convention:

```
atck_obj_attribute().
```

All attributes are read-only. The only way an ATtaCK tool can modify the target application is by writing instrumentation code into it, as covered in the next lesson.

Whenever one of these access methods returns a memory buffer (either a `const char*` or a `const void*`), that buffer is valid until the containing image object is closed. Buffers returned from the program object last until the program object is closed. In no case should you release these buffers yourself.

Let me apologize here in advance for the next two pages. There are a *lot* of attributes, and each attribute has an access method. The best way for me to tell you about these attributes is to tell you about their access methods, all eighty-four of them.

Right across this sea of text, though, is our first ATtaCK tool. Just hang in there and we'll get through this together.

## Program Attributes

Remember that the program object doesn't have an abbreviation, so instead of `atck_prog_attribute()`, these methods all follow the pattern `atck_attribute()`.

```
const char* atck_appname(atck_prog_t* self)
```

The application's filename...

```
const char* atck_iappname(atck_prog_t* self)
```

The name of the instrumented application file. When you called `atck_open()`, if you specified an output filename using its *outfile* argument, then this string will contain that argument, with any directory specifiers expanded into their real values. If you passed a `NULL` for *outfile*, then `NULL` is what you'll get back here.

```
atck_endian_t atck_byteorder(atck_prog_t* self)
```

You've already seen this attribute, the program's byte order. Its possible values are ATCK\_BE for big-endian (e.g., the Macintosh) or ATCK\_LE for little-endian (PC or PlayStation 2).

```
unsigned atck_nimg(atck_prog_t* self)
unsigned atck_nproc(atck_prog_t* self)
unsigned atck_nent(atck_prog_t* self)
```

The number of images in the program (always 1 for the PlayStation 2), and the number of procedures or entry points in the program across all images.

```
unsigned atck_nproc_rdable(atck_prog_t* self)
unsigned atck_nent_rdable(atck_prog_t* self)
unsigned atck_ncall_rdable(atck_prog_t* self)
unsigned atck_nbb_rdable(atck_prog_t* self)
unsigned atck_ninst_rdable(atck_prog_t* self)
```

These attributes contain the number of procedures, entry points, call sites, basic blocks and instructions in the entire program that are *readable*, e.g., that are not marked with the ATCK\_ATTR\_NOREAD attribute. This is discussed under Procedure Attributes, below.

```
unsigned atck_nproc_instr(atck_prog_t* self)
unsigned atck_nent_instr(atck_prog_t* self)
unsigned atck_ncall_instr(atck_prog_t* self)
unsigned atck_nbb_instr(atck_prog_t* self)
unsigned atck_ninst_instr(atck_prog_t* self)
```

The number of procedures, entry points, call sites, basic blocks and instructions in the entire program that are *instrumentable*, e.g., that are marked with the ATCK\_ATTR\_INSTR attribute. This is discussed under Procedure Attributes, below.

```
unsigned atck_nproc_skip(atck_prog_t* self)
unsigned atck_nent_skip(atck_prog_t* self)
unsigned atck_ncall_skip(atck_prog_t* self)
unsigned atck_nbb_skip(atck_prog_t* self)
unsigned atck_ninst_skip(atck_prog_t* self)
```

The number of procedures, entry points, call sites, basic blocks and instructions in the entire program that you have marked with the ATCK\_ATTR\_SKIP attribute. This is discussed under Procedure Attributes, below.

```
size_t atck_szdisbuf(atck_prog_t* self)
```

The size in bytes of the longest string that will ever be printed by atck\_inst\_dis() (the instruction disassembly method) for this program.

## Image Attributes

```
const char* atck_appname(atck_img_t* self)
```

The name of the image: An image might not have a file name, in which case this attribute is NULL.

```
unsigned atck_img_nproc(atck_img_t* self)
unsigned atck_img_nent(atck_img_t* self)
```

The number of procedures or entry points in the image.

```
unsigned atck_img_nproc_rdable(atck_img_t* self)
unsigned atck_img_nent_rdable(atck_img_t* self)
unsigned atck_img_ncall_rdable(atck_img_t* self)
unsigned atck_img_nbb_rdable(atck_img_t* self)
unsigned atck_img_ninst_rdable(atck_img_t* self)
```

The number of readable procedures, entry points, call sites, basic blocks and instructions in the image.

```
unsigned atck_img_nproc_instr(atck_img_t* self)
unsigned atck_img_nent_instr(atck_img_t* self)
unsigned atck_img_ncall_instr(atck_img_t* self)
unsigned atck_img_nbb_instr(atck_img_t* self)
unsigned atck_img_ninst_instr(atck_img_t* self)
```

The number of instrumentable procedures, entry points, call sites, basic blocks and instructions in the image.

```
unsigned atck_img_nproc_skip(atck_img_t* self)
unsigned atck_img_nent_skip(atck_img_t* self)
unsigned atck_img_ncall_skip(atck_img_t* self)
unsigned atck_img_nbb_skip(atck_img_t* self)
unsigned atck_img_ninst_skip(atck_img_t* self)
```

The number of skipped procedures, entry points, call sites, basic blocks and instructions in the image.

## Procedure Attributes

```
atck_attr_t atck_proc_attr(atck_proc_t* self)
```

This attribute indicates the procedure's ability to be read and instrumented. There are four possible values:

- **ATCK\_ATTR\_INSTR** indicates that the procedure can be instrumented. Almost every procedure will have this attribute.
- **ATCK\_ATTR\_SKIP** indicates that you have flagged the procedure to be skipped. This attribute is set by your own code, using the functions described in the notes. It's essentially just a reminder to yourself—ATtaCK treats **ATCK\_ATTR\_SKIP** exactly the same as **ATCK\_ATTR\_INSTR**.
- **ATCK\_ATTR\_RDONLY** indicates that ATtaCK is unable to instrument the procedure, although it can still navigate through the code. This would happen

with any procedures that weren't compiled by CodeWarrior—every procedure in the PlayStation 2 SDK, for instance. (Sorry!)

- **ATCK\_ATTR\_NOREAD** indicates that ATtACK is unable to read the procedure's code, which means ATtACK also cannot instrument the procedure. This will only be the case if the procedure contains invalid machine code, in which case something very strange is happening.

```
const char* atck_proc_name(atck_proc_t* self)
```

---

## Marking Code to Skip

```
void atck_skipimg(atck_prog_t* self, const char* imagename)
void atck_skipfile(atck_prog_t* self, const char* filename)
void atck_skipproc(atck_prog_t* self, const char* procname)
```

These three routines allow you to mark procedures with **ATCK\_ATTR\_SKIP**. As you iterate, you can check for this attribute and skip the procedures you marked. ATtACK in no way enforces the skip attribute—it's just a flag provided for your convenience.

Few tools need these functions. If you think they might be useful, refer to the [documentation](#).

---

The name of the procedure, if it has one; **NULL** otherwise. Remember that C++ function names will usually be mangled, and it's the tool's responsibility to unmangle them.

```
atck_addr_t atck_proc_addr(atck_proc_t* self)
```

The lowest address of the procedure. This is the address of the first basic block in the procedure, not necessarily the address of the first entry point (although for C or C++, those will almost always be one and the same).

```
const char* atck_proc_file(atck_proc_t* self)
unsigned atck_proc_line(atck_proc_t* self)
```

The source file and line number of the first line of the procedure. If this information is not available, the filename will be **NULL** and/or the line number zero.

```
unsigned atck_proc_nent(atck_proc_t* self)
unsigned atck_proc_ncall(atck_proc_t* self)
unsigned atck_proc_nbb(atck_proc_t* self)
unsigned atck_proc_ninst(atck_proc_t* self)
```

The number of entry points, call sites, basic blocks and instructions in the procedure. For unreadable procedures, the number of call sites, basic blocks and instructions will be zero.

## Entry-Point Attributes

```
const char* atck_ent_name(atck_ent_t* self)
```



The name of the entry point, if it has one; `NULL` otherwise. Entry point names corresponding to C++ functions will require unmangling.

```
atck_addr_t atck_ent_addr(atck_ent_t* self)
```

The address of the entry point.

```
const char* atck_ent_file(atck_ent_t* self)
unsigned atck_ent_line(atck_ent_t* self)
```

The source file and line number of the entry point. If this information is not available, the filename will be `NULL` and/or the line number zero.

## Call-Site Attributes

```
atck_bool_t atck_call_istargknown(atck_call_t* self)
```

`ATCK_TRUE` if the target of this call site is known—that is, if this is a static call.

`ATCK_FALSE` if the target is dynamically called via a register or variable.

```
const char* atck_call_targname(atck_call_t* self)
```

The name of the target procedure, if the target is known and if it has a name; `NULL` otherwise.

```
atck_addr_t atck_call_targaddr(atck_call_t* self)
```

The address of the target procedure, if known, zero if not.

```
atck_attr_t atck_call_targattr(atck_call_t* self)
```

The target procedure's readability attribute, or `ATCK_ATTR_NOREAD` if the target address isn't known or lies outside the application.

```
atck_bool_t atck_call_returns(atck_call_t* self)
```

`ATCK_TRUE` if the call appears to return to the caller when the called procedure finishes; `ATCK_FALSE` otherwise. Call sites usually return to the caller, but some call sites do not due to compiler optimizations or hand-coded assembly techniques.

```
atck_addr_t atck_call_addr(atck_call_t* self)
```

The address of the call site, rather than the address of the target.

```
atck_bool_t atck_call_iscond(atck_call_t* self)
```

`ATCK_TRUE` if the call is conditional, `ATCK_FALSE` if it is unconditional.

## Basic-Block Attributes

```
atck_addr_t atck_bb_addr(atck_bb_t* self)
```

The starting address of the basic block.

```
unsigned atck_bb_ninst(atck_bb_t* self)
```

The number of instructions contained within the basic block.

```
atck_attr_t atck_bb_attr(atck_bb_t* self)
```

The readability attribute of the basic block's parent procedure.

```
const void* atck_bb_raw(atck_bb_t* self)
size_t atck_bb_rawsize(atck_bb_t* self)
```

These methods expose the raw machine code contained with the basic block. `atck_bb_raw()` returns a buffer containing the raw bytes, in the byte ordering of the target processor; `atck_bb_rawsize()` returns the size in bytes of this buffer.

## Instruction Attributes

This is the last section! Unfortunately, it's also the longest section—the instruction object, naturally enough, has the most interesting attributes and requires the most explanation.

```
atck_addr_t atck_inst_addr(atck_inst_t* self)
```

The starting address of the instruction.

```
const char* atck_inst_file(atck_inst_t* self)
unsigned atck_inst_line(atck_inst_t* self)
```

The source file and line number of the instruction. If this information is not available, the filename will be `NULL` and/or the line number zero.

```
const void* atck_inst_raw(atck_inst_t* self)
size_t atck_inst_rawsize(atck_inst_t* self)
```

These methods expose the raw machine code for the instruction. `atck_inst_raw()` returns a buffer containing the raw bytes, in the byte ordering of the target processor; `atck_inst_rawsize()` returns the size in bytes of this buffer.

```
atck_op_t atck_inst_op(atck_inst_t* self)
```

The instruction's "pseudo-opcode." A pseudo-opcode identifies the function performed by an instruction, but may not correspond to an actual hardware opcode for the target processor.

A table listing the pseudo-opcode of every EE instruction starts on page ATK-231 of the online documentation.

```
size_t atck_inst_dis(atck_inst_t* self, char* buffer)
```

Writes a textual disassembly of the instruction to *buffer*, a string buffer you must allocate yourself. In order to avoid overflow, *buffer* must be at least `atck_szdisbuf()` bytes long. This routine returns the number of bytes written to the buffer, excluding the null terminator.

This disassembly is complete, but not symbolic or particularly friendly. Here's a brief sample:

```
addu      r4, r0, 325
lui       r2, 4096      ! %hi(0x10000000)
or        r3, r2, 36864 ! %lo(0x9000)
sw        r4, 0(r3)
paddub    r4, r0, r0
paddub    r5, r0, r0
```

```
jal          0x103738
nop
```

You could take the disassembly results and run them back through an assembler to get the program again, but if you want something usefully human-readable, you'll probably need to do some back-end processing. We'll see one way to do that in the example program at the end of this lesson.

## Instruction Classification

These attributes provide a generic, cross-platform classification of the instruction. They each return `ATCK_TRUE` if the instruction performs the specified function, or `ATCK_FALSE` otherwise. These classifications are not disjoint—a single instruction may belong to more than one class. For example, the MIPS BC1 instructions are conditional branches that touch the floating-point unit.

A table showing the classification of every EE instruction starts on page ATK-231 of the online documentation.

```
atck_bool_t atck_inst_isload(atck_inst_t* self)
```

Does the instruction load a value from memory?

```
atck_bool_t atck_inst_isustore(atck_inst_t* self)
```

Does the instruction unconditionally store a value to memory?

```
atck_bool_t atck_inst_iscstore(atck_inst_t* self)
```

Does the instruction conditionally store a value to memory?

```
atck_bool_t atck_inst_isubbranch(atck_inst_t* self)
```

Does the instruction unconditionally branch (that is, change the control flow of the application)?

```
atck_bool_t atck_inst_iscbranch(atck_inst_t* self)
```

Does the instruction conditionally branch?

```
atck_bool_t atck_inst_isbranch(atck_inst_t* self)
```

Does the instruction branch conditionally *or* unconditionally?

```
atck_bool_t atck_inst_isfp(atck_inst_t* self)
```

Does the instruction touch the floating-point unit?

## Register Usage

```
void atck_inst_inregs(atck_inst_t* self, atck_regs_t* regs)
```

```
void atck_inst_outregs(atck_inst_t* self, atck_regs_t* regs)
```

These functions identify which registers the instruction reads and modifies, respectively. They're very important, but covering them would take a page all by itself. We've already slogged through enough API information, so let's move on to looking at some code to try out our new knowledge. We'll come back to register usage in Lesson 05.

## Okay, You Can Wake Up Now...

...we're done with the API documentation for this lesson. Go on to the next section for the sample tool I keep promising.

## *A Static Analysis Tool*

### What is Static Analysis?

Dynamic analysis is examining a running program, using a debugger or a profiler. Static analysis is examining a program's code without running it, by simulating the behavior of the CPU. For example, you're able to step from a call instruction into the called procedure because you (or rather, ATtaCK) know how to identify an instruction and how the CPU executes it. So in many ways, static analysis can tell you *everything* about your program. Indeed, if your CPU simulator modeled every effect of every instruction, there'd be no difference at all between static and dynamic analysis apart from speed.

Sorry, ATtaCK is smart, but not *that* smart. Its CPU simulator is only concerned with the most important aspects of how an instruction is executed:

- Whether it branches, and to where
- Whether it calls a procedure, and which one it calls
- Whether it loads from or stores to memory
- Which registers it reads and writes
- Whether its execution is conditional or unconditional

So while ATtaCK can't replace your T10000 system, you can still perform some very valuable analysis tasks without ever running the target application.

For one thing, you can selectively disassemble your program and display the result. Don't underestimate how powerful a debugging tool Eyeball v1.0 is. If you understand assembly code, you can catch a lot of problems in your application just by scanning it for sections that "don't look right." This is especially true in C++, where a quick look at the actual assembly code for a function can often reveal that hidden temporary variable whose constructor and destructor are killing your performance.

Static analysis really shines for analysis tasks that are either too slow or have too many side effects to perform on the target system. Disassembly is a perfect example of that: You can't read your application at the speed the CPU executes it, and your application can't effectively run at the speed that you read. Disassembly must be done statically. A task like inline optimization, where you look at the program as a whole to evaluate which functions to inline, also consume too much time to perform against a running application.

Another task for which static analysis works well is code validation. For example, there are certain OS routines that a shipping PlayStation 2 application must never

call. You can use static analysis to sweep your code and ensure that you never call those routines. We'll work with an extremely valuable code validation tool in Lesson 06 that does exactly that.

For now, though, we're going to look at the simplest "real" ATtACK tool to write: a disassembler.

## Display.c: A Simple Disassembler

### Creating the Project

Your ATtACK installation already includes a working project for `display.c`, under `Examples\Display`. Since we're not going to modify `display.c`, you can save yourself some time and just use the example project.

However, if you want to make changes to `display.c` later on, it's best to create a new project. In Lesson 01, we went through the steps of creating a new ATtACK tool that includes both instrumentation and analysis code. If you're just writing an instrumentation tool, you can take a shortcut:

- Use "Win32 C Stationery" to create a new Win32 C console application.
- Add the `atck-ps2\include` folder to the project's paths.
- Add `atck.lib` to the project's files.
- Write your program. In this case, you would copy `display.c` into your project folder, add the copy to the project and remove the blank `main.c` created by the stationery.

### Testing the Tool

Once you've either opened or created the project for `display.c`, hit F7 to make it. The example project is already made; if you want to see it build just to reassure yourself, hit ctrl-minus to remove object code, then make the program. If you get errors during the make, see Lesson 01 for troubleshooting.

This is a console application, so open a command console and go to the directory containing the program. For the example project that came with ATtACK, the program is `Examples\Display\bin\ps2\display.exe`. For a project created using the CodeWarrior stationery, the program is in the `Bin` subdirectory and is named `noname.exe` unless you gave it a different name at the "x86 Target" settings panel.

Run the program once with no command-line just to make sure it works. If you get a "cannot find atck.dll" dialog box, you need to modify your path; see Lesson 01 for troubleshooting. Otherwise, you should see a line like this:

```
Usage: display <app>
```

The only command-line option is the name of the program itself. Now, this tool will disassemble the *entire* program, so we need to be careful here. If you just take a normal program and run the tool, the output will scroll past for about five to ten minutes—literally! Even a small sample application—such as my personal

favorite, Blow, found in the Examples\sce200\vu1\blow directory—will go on forever. When you get back from getting yourself a cup of coffee, here's what you'll see:

```

0x001164d8 0x27bdfdf0 addu    r29, r29, -48
0x001164dc 0xffbf0020 sd      r31, 32(r29)
0x001164e0 0x27a40010 addu    r4, r29, 16
0x001164e4 0xe7ac0010 swc1   f12, 16(r29)
0x001164e8 0xc045692 jal     0x115a48
npack_f'
0x001164ec 0x03a0282d daddu   r5, r29, r0
0x001164f0 0x8fa7000c lw      r7, 12(r29)
0x001164f4 0x8fa40000 lw      r4, 0(r29)
0x001164f8 0x0007383c dsll32  r7, r7, 0
0x001164fc 0x8fa50004 lw      r5, 4(r29)
0x00116500 0x8fa60008 lw      r6, 8(r29)
0x00116504 0xc04562c jal     0x1158b0
ake_dp'
0x00116508 0x000738ba dsrl    r7, r7, 2
0x0011650c 0xdfbf0020 ld      r31, 32(r29)
0x00116510 0x03e00008 ret
known
0x00116514 0x27bd0030 addu    r29, r29, 48

```

Fig. 02-01: Sample Output of display.exe

The better option is to redirect the output to a text file. This runs *much* faster, and the results are actually readable. Type the following, all as one line:

```
display "C:\Program
Files\Metrowerks\CodeWarrior\Examples\sce200\vu1\blow\blow.elf
" > output.txt
```

(Don't leave out the quotes—they're important!) Now all that output goes to the text file, which you can open in Notepad.

If output.txt is empty or contains an error message, it's probably because your copy of ATtACK isn't installed properly, or perhaps because you don't have the PlayStation 2 SDK installed. Go back to Lesson 01 for troubleshooting.

## How It Works

Okay, you're still here, so I'll assume the program worked. Now let's look at the tool's structure. Open up display.c in your IDE. As you can see, this is a simple command-line tool, with just one source file, display.c. main() does all the driving. The initialization code at the start of main(), and the shutdown code at the end, should look very familiar. This boilerplate code isn't going to change much from tool to tool, so get used to it.

main() calls Display() to do the actual processing. Display() in turn calls PrintProc(), PrintEnts(), PrintCalls() and PrintBB() to step down into the target application. We'll look at those functions in the next section.

## Navigation and "Analysis"

### The Display() Function

The first thing `Display()` does is declare all the variables it's going to need. Since it declares image and procedure iterators, and image and procedure handles, we can already get a good sense of what `Display()` is going to do.

Next, `Display()` gets the byte order of the target, even though we already know it's little-endian. It also allocates a string buffer by using `atck_malloc()`, which is just a wrapper for `malloc()`; ATtACK provides it so your code can rely on a specific memory allocation routine across all platforms. This buffer will be used to display disassembled instructions, so we have to make sure it's `atck_szdisbuf()` bytes long.

Demonstrating the attribute access methods perfectly, `Display()` now proceeds to dump out information about the target program, listing the number of images, procedures, entry points, call sites, basic blocks and instructions the program contains.

Next, it allocates an image iterator and starts walking through the image list to generate an image-level summary of the same information. For each image in turn, `Display()` allocates a procedure iterator to step through and display the procedure-level statistics.

The procedure iterator loop keeps going until the next method returns `NULL`. The image loop then releases the procedure iterator. Remember that it's your job to free up iterators, since they have no parents to clean up after them. There's no way to reinitialize or retarget an iterator; you just have to release it and create a new one.

The image iterator loop then releases the image handle it got back. Notice that it *doesn't* release any of the procedure handles; they are owned by and released by the parent image. It could just let the program object release the image objects at the end of the run, but images can be very large, so releasing each image once it's no longer required can save a lot of memory. Of course, PlayStation 2 programs only have one image each, so the question here is moot, but this is good practice for the future.

When the image iterator's `_next()` method returns `NULL`, the tool has finished gathering the summary statistics. Now it iterates across the images all over again, to disassemble them. To do this, it must release the old image iterator and create a new one. Why not just jump back to the beginning of the iterator using `atck_imgit_first()`? Because running through the iterator again would just give back the same handles that were already released.

The second image iterator loop is much like the first. Instead of printing summary information for each procedure in the image, this loop calls `PrintProc()`, passing it the disassembly buffer allocated earlier. As the tool finishes with each image, it releases the image for the last time. When the loop ends, the tool releases this second image iterator and returns to `main()` for final cleanup.

## The Remaining Functions

`PrintProc()`, `PrintEnts()` and `PrintCalls()` are not particularly enlightening: They create an iterator, run through every object in the list to display its attributes, then release the iterator. These functions take the same principles used in `Display()` and apply them to procedures, entry points and call sites. By this point, though, you should be very confident in your iteration skills, so we'll move on to something more interesting: `PrintBB()`, where the actual disassembly happens.

After declaring variables, `PrintBB()` immediately does something that may look strange. It gets the number of instructions in the basic block, and the total number of raw bytes those instructions take up, and divides the latter by `sizeof(atack_uint32_t)` to make sure that it equals the former. In the disassembly loop later on, casting them to an unsigned 32-bit int prints out each instruction's raw bytes. That's safe when working with the PlayStation 2's MIPS instruction set, where each instruction is exactly 32 bits, but if that's not what we're working with, we want to know about it now!

Obviously this code only works when dealing with MIPS and similar RISC architectures. This is a perfect example of when to violate cross-platform compatibility. ATtaCK tools are so simple to write that you can afford to create throwaway code that only works for one platform (or even for one target application!). You've got the *opportunity* to write reusable tools, but don't hesitate to just slap together a one-shot solution for a problem at hand.

Moving onward, the next code should be very familiar by now. It creates an iterator to traverse the instruction list. The `ATCK_FLAGS_ITLIFE` flag indicates that the instruction objects created should only live as long as the iterator itself should. That's almost always what you want to use—without that flag, instruction objects can wind up consuming a *lot* of memory.

The loop then prints out five pieces of information for each instruction: its source file name and line number, its address, its raw bytes, and its disassembly string. The `ATCK_SWAPTOUN32()` macro conditionally byte-swaps the instruction's raw bytes if the host platform has a different byte order than the target. Thus, while this tool will only work for the PlayStation 2, it can be run on multiple host platforms.

After the disassembly information that's common to all instructions, the loop prints out the instruction's classifications, if any. If the instruction is a branch, the address is printed. If the instruction is a call, the target symbolic name (if known) or address is printed. Notice how the tool gets the instruction's corresponding call-site object then uses that object to get the call's target name and address.

When the loop finishes, the instruction iterator is freed and the function returns.

## Future Improvements

And that's the end of our first "real" ATtaCK tool. Its biggest problem is the simplistic output routines. Dumping to `stdout` is not really appropriate for this tool, since any but the shortest application is going to generate tens of thousands



of lines of output text. A more user-friendly approach would be to save the output to a text file, printing an ongoing progress report (such as a string of periods) to the console. Better still, save the output into a format more useful than just plain text.

Even then, the resulting disassembly file will be hard to work with. It'd be easy to add command-line options that let the user specify starting and ending points for the disassembly—from address to address, from source line to source line, or by function name. Once the scope of the disassembly is limited, console output becomes more manageable, so the user should get the option to switch between console and file output.

## **So Far So Good!**

Well, you'll be happy to know that this was the longest and most content-rich lesson of the entire course. If you look at the ATtaCK docs, you'll see that we just covered half the manual! That doesn't mean we're halfway done with the course, but it does mean we've gotten half of the boring API details out of the way. Future lessons will spend less time on type definitions and function arguments and more time on the theory and practice of ATtaCK analysis. In fact, the last two lessons are nothing *but* theory and practice.

For now, though, we've still got some more API to cover, so it's on to Lesson 03: Instrumenting an Application.

## Lesson 02 Assignment

Write the following function:

```
atck_inst_t* GetNextInstruction(atck_inst_t* cur);
```

This function should return a handle to the instruction that will get executed after the current one. If that cannot be determined, return NULL.

Hint:

The key here is to understand that the next instruction by address won't necessarily be the next instruction executed. There are a few situations to deal with:

- If the current instruction isn't the last one in a basic block, then the next instruction is simply the next instruction returned from the block's iterator.
- If the current instruction is the last one in the block and is an unconditional branch, the next instruction will be the target of the branch. (If it's a conditional branch, you should just return NULL, since by definition you don't know which instruction will be executed next—you know which of two possible ones will be executed, but that's not what you were asked to find out!)
- If the current instruction is the last one in the block and is not a branch, then it may be in the load-delay slot of the previous instruction. In that case, the next instruction will be the target of the previous branch instruction.
- If the current instruction falls at the end of a basic block but isn't a branch and isn't in a load-delay slot, then the next instruction is just the first instruction of the next basic block.

Answer:

The file ex02-01.c, which you can get from the "supplemental material" folder, performs all the required checks. Note that it uses "image-life" instruction handles, which are wasteful. However, it has to release its instruction iterators before returning in order to avoid memory leaks, and so iterator-life instruction handles would be invalid by the time the caller received the return value. For extra credit, figure out how to use iterator-life handles with this code!

## Lesson 02 Quiz

1. How many instructions are in each basic block?
  - A One
  - B Two, because the EE loads two instructions at once
  - C Sixteen, because the EE's instruction cache lines are 64 bytes wide
  - D None of the above
2. When ATtaCK iterates through all the procedures in an image, which of the following will appear in the list?
  - A Procedures that do not have symbolic debugging information
  - B Procedures that are never called
  - C Both
  - D Neither
3. Which of the following is most likely to be a legitimate ATtaCK function?
  - A `open_inst( atck_inst_t`
  - B `atck_iprog_close( atck_iprog_t*)`
  - C `atck_get_proc_size( atck_proc_t*)`
  - D `atck_bb_findnext( atck_proc_t*)`
4. You have two instruction-object handles, X and Y. Comparing them, you learn that X is less than Y. What do you now know about these two instructions?
  - A Nothing
  - B X was created before Y
  - C X has a lower address than Y
  - D X and Y represent two different instructions
5. True or false: You cannot write ATtaCK instrumentation tools in C++.
  - A True
  - B False
6. You can use a procedure handle to create an iterator for three of these object types. Which one can *not* be iterated across in a procedure?

- A Instructions
  - B Basic blocks
  - C Entry points
  - D Call sites
7. True or false: The last instruction in a basic block will always be some kind of branch, call or return.
- A True
  - B False
8. Which of the following objects needs to be released after use?
- A Program
  - B Image
  - C Procedure
  - D Instruction
9. Of the following, which can appear multiple times within a basic block?
- A Branches
  - B Entry points
  - C Call sites
  - D None of the above
10. True or false: Before ending your ATtaCK session, you must free any strings returned from methods like `atk_appname()`.
- A True
  - B False

## Lesson 03: Instrumenting an Application

---

The key to run-time analysis with ATtaCK is instrumentation: inserting analysis calls into your application's binary image. In our third lesson, we'll cover instrumentation in depth. We'll also spend some time looking at simple analysis code. Finally, we'll examine ProcCount, a basic procedure counter.

### *Instrumentation Concepts*

#### Lesson Objectives

In this lesson, we'll cover two more steps of the ATtaCK process: adding instrumentation calls and writing out the application. Writing out the application is easy, and will take a small part of one page. Learning how to design, declare, build and invoke analysis code will take the rest of this lesson and the bulk of Lessons 05, 06 and 07. So don't feel like you have to become an expert on analysis code right away... we'll visit this issue again later.

On the other hand, there *will* be a quiz at the end of this lesson, so pay attention!

#### Instrumentation and Analysis

You're probably used to working with a sampling profiler. Such a tool runs alongside the target application, halting it many thousands of times a second to read the contents of the program counter (or instruction pointer in Intel-land). This lets the compiler build a statistical picture of the application's performance: If the program counter was within the `RenderWorld()` function 72,600 times out of the 100,000 times the profiler stopped the program, then it's a good bet that `RenderWorld()` probably takes up 72.6% of the application's clock cycles.

It's a good bet, but not a *sure* bet. Sampling profilers are not perfectly accurate—very fast function calls might fall in between samples, and line-by-line performance results within a procedure aren't very reliable at all.

To improve accuracy, the profiler can take more samples per second, but that slows down the target application. When working with a game, sampling profilers can slow the code down so much that the game becomes unplayable. Profiling then has to be a separate task from debugging or playtesting, and the profiler's results don't represent typical gameplay.

Instrumentation takes a different approach. Instead of halting the target application thousands of times a second to read its state, instrumenting profilers add code to the target application that log events when the application's state changes. For instance, an instrumenting profiler might add code to the start of every function to log that that function got called.

Notice how instrumentation solves the problems of sampling. First, instrumentation is 100% accurate—nothing falls through the cracks. Second, the code to log events usually runs much faster than halting the program to sample its

state externally. CPUs are much faster at function calls within a thread than at context switches between threads.

Best of all, an instrumenting profiler doesn't have to instrument the entire application. If you suspect that `RenderWorld()` is where the big time-sinks are, you can tell an instrumenting profiler to concentrate on that one function. A sampling profiler must run constantly if it's to be of any value at all.

## Binary Code Instrumentation

There are two ways to instrument an application, in source code and in binary code. You've almost certainly created source-level instrumentation yourself a time or two: When you write code in your display loop that gets the time at the start and end of a frame, and compares those times to generate a "frames per second" statistic, you're using source code instrumentation.

Source code instrumentation is easy but time-consuming. You have to write the code, add it to the correct places in the application, recompile and run the test, then go back, remove the code and recompile again to "shut off" the profiling.

To save you that effort, ATtaCK uses binary code instrumentation. In this technique, an instrumentation tool (e.g., an ATtaCK program) inserts instrumentation calls into your compiled executable. Instrumentation calls, conceptually, are function calls to routines that log events or monitor the application's state. Those routines are called analysis code.

For example, you might want to know how many times a particular function gets called. Your instrumentation tool would add an instrumentation call to the start of that function. The instrumentation call would call an analysis routine that incremented a counter. At the end of the run, the counter would contain the number of times the function had been called.

As you think about this, I'm sure you'll start to see how powerful ATtaCK's binary code instrumentation can be. Instrumentation calls can be placed before or after any entity, from individual instructions all the way up to the program as a whole. You can specify arguments for these instrumentation calls—either static arguments, such as an ID value for the procedure being instrumented, or dynamic arguments, such as the contents of a particular register. And the analysis code can do anything it wants with this information, including halting the application altogether.

Unfortunately, all this talk about the principles behind instrumentation and analysis makes it sound harder than it really is. To quell any fears, let's look briefly at an actual analysis routine:

```
void CountProc(atchk_uint32_t procID)
{
    ProcCounts[procID]++;
}
```

`ProcCounts[]` is an array of counters, one for each procedure. Procedures are identified by number using `procID`. To use this array, we add a call to

`CountProc()` at the start of each procedure, passing in an ID number unique to that procedure. When the program runs, these counters will get incremented. After the program finishes, we'll be able to look at each counter for each procedure.

Running the program and reading the array of counters are the subject of the next lesson, though we'll touch on them briefly when we examine a sample program later on. For now, let's look at how instrumentation calls work.

## ***Instrumentation Calls***

Analysis routines are C `void` functions that you write and compile into a temporary executable. ATtaCK will link this file later on to your target application, so that instrumentation calls inserted into the target can call to the analysis code you write.

Analysis routines don't return values, but they do accept arguments. Each instrumentation call is thus a function call that passes a list of arguments but does not expect a return value. Since the instrumentation call is inserted into the target application by your instrumentation tool, the upshot of all this is that your instrumentation tool passes the arguments to your analysis code.

## **Passing Arguments**

Let's take a quick look at how the instrumentation tool "calls" the analysis code:

```
iproc = 0;
pproc = atck_procit_first(ppi);
while (pproc)
{
    atck_proc_callbefore(pproc, pproto, iproc++);
    pproc = atck_procit_next(ppi);
}
```

You should immediately recognize everything in this code fragment except for the fourth line. If you don't, go back to Lesson 02 and review that now, because believe me, this isn't going to get any easier from here.

We're not going to go into much detail on `atck_proc_callbefore()` right now. This is the ATtaCK function used to insert an instrumentation call to a particular analysis routine into the start of a target procedure. The function is a method of the procedure object, the "self" handle of which is passed as the first argument.

The analysis routine is represented by a "prototype" object, which stores ATtaCK's definition of the routine (including its expected arguments). A handle to that object is passed as the second argument; in this example, `pproto` is a handle to the prototype for the `CountProc()` function that we've already seen.

The remaining arguments of `atck_proc_callbefore()` are the argument list to be passed to the analysis routine. In this case there's just one argument, `iproc`. As

you can see, this is a counter that gets incremented for each procedure, so it will range from 0 to the total number of procedures covered by the iterator minus 1.

Every time `atk_proc_callbefore()` is called, ATtaCK inserts a call to `ProcCount()` right at the start of the target procedure, passing it the value of `iproc`. If the target's `main()` function happens to be the 17<sup>th</sup> function returned by the iterator, the result would be something like this:

```
int main(int argc, char** argv)
{
    ProcCount(16);

    /* REST OF ORIGINAL MAIN() GOES HERE */

    ...
}
```

Okay, now before I get into trouble here, let me stress that ATtaCK uses *binary code* instrumentation. No source code file gets modified, and the program doesn't get compiled again! I'm just describing it here *as if* the source code were instrumented, in order to make it clear what's going on. Conceptually, ATtaCK inserts the line `ProcCount(16);` at the top of `main()`. That line is the instrumentation call, and `ProcCount()` is the analysis routine.

I suppose I could have shown an assembly version, something like this:

```
main:
    xor $a0, $a0, $a0      # zero-out register $a0
    addi $a0, $a0, 16      # put the value 16 into register
$a0
    jal ProcCount          # call ProcCount
    # rest of original main() goes here
    ...
```

... except that it might or might not look like that. The instrumentation call is *conceptually* a function call, but in practice, ATtaCK is free to perform the task however it sees fit. In this case, since `ProcCount()` is so simple, ATtaCK would almost certainly inline it, inserting the entire body of `ProcCount()` into the beginning of `main()`.

The point of all this is to demonstrate that the third and subsequent arguments passed into the instrumentation method `atk_proc_callbefore()` *become* the first and subsequent arguments to the analysis routine. The best way to envision this process is that your instrumentation tool is passing values directly to your analysis code, using the target application as a middleman.

## Static Arguments vs. Dynamic Arguments

The `iproc` counter in this example is a *static* argument: Although the instrumentation code runs through many values for `iproc`, each individual instrumentation call that gets inserted into the program treats the value as a constant. The instrumentation call at the top of `main()` will *always* pass 16 to `ProcCount()`.



You can also specify *dynamic* arguments. With a dynamic argument, your instrumentation tool doesn't know the value. Instead, you specify which aspect of the target application should be passed at runtime. ATtaCK then creates an instrumentation call that reads that value and passes it on to the analysis routine.

For example, if you write an analysis routine to keep track of how deep the stack has grown, you'd want to pass the value of register 29, the stack pointer. Obviously your instrumentation tool doesn't know that value. Instead, you pass a special code meaning "the value of register 29" to the instrumentation method.

ATtaCK then inserts instrumentation code that fetches the value of register 29, stores it in a register and *then* calls the analysis routine. The analysis routine itself, however, just sees an unsigned 32-bit integer coming in; it doesn't have to know anything at all about how registers or the stack work.

## ***Instrumenting an Application***

### **Instrumentation Methods**

#### **Common Characteristics**

Instrumentation calls get added to existing objects in the target application, such as instructions or procedures. Consistent with the rest of the ATtaCK API, this means that the functions to insert instrumentation calls are methods of code objects. As methods, these functions each take as their first argument a handle to the code object in question.

The second argument of an instrumentation method is always a handle to the target analysis routine. Specific analysis routines are represented by the prototype object, `atck_proto_t`. This object contains the routine's symbolic name and its argument list. For every analysis routine you create in C, you have to declare it in ATtaCK. The declaration function gives you back a handle to a prototype object, which you then pass to the instrumentation methods. This process will be covered more later.

The remaining arguments of an instrumentation method are the values that the instrumentation call should pass to the analysis routine.

Instrumentation methods follow the normal ATtaCK naming convention: `atck_obj_action()`. In this case, the action is either `callbefore`—add the instrumentation call before the specified object executes—or `callafter`—add the call after the object executes.

None of these methods returns a value. If the specified object cannot be instrumented, or if the specified analysis function is not appropriate for the object, ATtaCK prints a diagnostic message but does not generate an error code.

#### **Method Definitions**

```
void atck_callbefore(atck_prog_t* self, atck_proto_t* targ,
...)
```

```
void atck_callafter(atck_prog_t* self, atck_proto_t* targ,
...)
```

These methods add an instrumentation call at the start or end of the specified program. Remember that the program object has no abbreviation, so the method is `atck_callbefore()` rather than `atck_prog_callbefore()`.

```
void atck_img_callbefore(atck_img_t* self, atck_proto_t* targ,
...)
void atck_img_callafter(atck_img_t* self, atck_proto_t* targ,
...)
```

These methods add an instrumentation call at the start or end of the specified image.

```
void atck_proc_callbefore(atck_proc_t* self, atck_proto_t* targ,
...)
void atck_proc_callafter(atck_proc_t* self, atck_proto_t* targ,
...)
```

These methods add an instrumentation call at the start or end of the specified procedure. Procedures marked “unreadable” or “unwritable” cannot be instrumented, and ATtaCK will generate a diagnostic message if you try.

```
void atck_ent_callbefore(atck_ent_t* self, atck_proto_t* targ,
...)
```

This method adds an instrumentation call before the specified entry point—that is, before any code in the entered procedure executes. There is no such thing as instrumentation “after” an entry point. Entry points in procedures marked “unreadable” or “unwritable” cannot be instrumented, and ATtaCK will generate a diagnostic message if you try.

```
void atck_call_callbefore(atck_call_t* self, atck_proto_t*
targ,...)
void atck_call_callafter(atck_call_t* self, atck_proto_t*
targ,...)
```

These methods add an instrumentation call before or after the specified call site. Call sites *in* procedures marked “unreadable” or “unwritable” cannot be instrumented, and ATtaCK will generate a diagnostic message if you try. Call sites that *target* uninstrumentable procedures may themselves be instrumented, however.

```
void atck_bb_callbefore(atck_bb_t* self, atck_proto_t*
targ,...)
void atck_bb_callafter(atck_bb_t* self, atck_proto_t*
targ,...)
void atck_inst_callbefore(atck_inst_t* self, atck_proto_t*
targ,...)
void atck_inst_callafter(atck_inst_t* self, atck_proto_t*
targ,...)
```

These methods add an instrumentation call at the start or end of the specified basic block or instruction. Basic blocks and instructions in procedures marked

“unreadable” or “unwritable” cannot be instrumented, and ATtaCK will generate a diagnostic message if you try.

---

## Wot, No Filename?

Somewhat surprisingly, neither `atck_img_write()` nor `atck_finish_write()` accepts a filename to write the new executable to. They also don’t modify the original executable. So where does the filename come from? From the original `atck_open()` call.

The `atck_open()` function’s *outfile* argument specifies a *filename template* for use in writing out the executable. Since an application might have more than one image, the template allows you to specify a naming scheme by which new image filenames can be generated as necessary.

The format for this template is covered in the documentation on page [FIXUP: “ATCK Framework API”, page 13]. However, here’s a boilerplate definition that will almost always work for you:

```
"<dir><base>_toolname<suf>"
```

This template, which is used by all the example programs, places the new executable in the same directory as the original one, adding “\_toolname” to the end of the base name but keeping the file extension the same. Replace *toolname* with your tool’s name, such as `proccount`.

If you intend to download and execute the instrumented program immediately, and are content to throw the program away once the profiling session is done, you may specify `NULL` for the output filename. ATtaCK will then use temporary files to hold the instrumented program.

---

## Writing the Instrumented Application

```
atck_bool_t atck_img_write(atck_img_t* self)
```

If you add any instrumentation to an image, including to children of an image, you must tell ATtaCK to write out that image, using `atck_img_write()`, rather than simply releasing the image using `atck_img_release()`.

If you instrument some images in a program but not others, you only need to call `atck_img_write()` for the instrumented ones; the rest can be released normally. You *can* call it for an uninstrumented image, in which case the new image file is identical to the original one.

This method generates a name for the new image file using the filename template specified in the `atck_open()` call, then writes the image and releases all resources associated with the image. Do not attempt to call `atck_img_write()` more than once for a given image, since each call will try to write to the same file.

This method returns `ATCK_TRUE` if the image was written successfully, or `ATCK_FALSE` otherwise (for example, due to lack of disk space).

```
atck_iprog_t* atck_finish_write(atck_prog_t* self)
```

If you call `atck_img_write()` for any image in the program, call this method after you've finished writing the last image. It finalizes any instrumentation calls added to the program object, and then generates the new executable file using the instrumented images. It returns a handle to an instrumented program ready for downloading and execution, a topic covered in the next lesson.

## ***Instrumentation Sequencing***

Previously, I threw around the terms “before” and “after” pretty loosely. For simple tools, for instance, only those that instrument procedure, the intuitive definitions of “before” and “after” are good enough. For anything more complicated, you’ll want to know *exactly* where the instrumentation calls get placed.

The guiding principle for the sequencing of “before” calls is *top-down, then first in, first out*. If you add two instrumentation calls before the same object, the first one will be executed before the second one. But if you add two instrumentation calls in the same location via two separate objects, the call added to the “higher” object executes before the call added to the “lower” object, regardless of the order in which they were added.

For example, instrumentation added to the start of a basic block always executes before instrumentation added to the start of the first procedure in the block. Two calls added before the same instruction, however, execute in the order they were added.

For “after” calls, the principle is *bottom-up, then first in, first out*. If you add two instrumentation calls after the same object, the first one will be executed before the second one, just as if they were added before the object. But if you add two instrumentation calls after the same location via two separate objects, the call added to the “higher” object executes after the call added to the “lower” object, regardless of the order in which they were added.

Thus, instrumentation added to the end of a procedure always executes after instrumentation added to the end of the last basic block in that procedure. Two calls added to the end of the same basic block, however, execute in the order they were added, *not* the reverse order.

## **Programs**

Instrumentation added before the program object is called before the first instruction in the application, and indeed before any instrumentation added with any other call. Instrumentation added after the program object is called after the last instruction in the application and after instrumentation added with any other call.

Program instrumentation is the perfect place for any initialization or shutdown functions your analysis code needs. For example, if your analysis code uses a memory buffer, you should probably allocate and initialize it in a function called via `atck_callbefore()`, then release it from an `atck_callafter()` function.

## Images

Instrumentation added before an image object is called after the image is loaded but before the first instruction in that image executes, and before any instrumentation associated with any child of that image. Instrumentation after the image executes immediately before the image is unloaded, and thus after any instrumentation associated with any child object.

If a program has only one image, or if it has one main image that remains in memory throughout execution, that image will still get “unloaded” when the program exits. In that case, instrumentation after the image will execute just before any instrumentation added after the program object.

## Procedures

Instrumentation added before a procedure is called just after the procedure is entered, but before any instructions in the procedure are executed. The same instrumentation call is executed regardless of which entry point the application uses to enter the procedure.

Likewise, instrumentation added after a procedure is called just before the procedure returns to its caller, and the same instrumentation is executed no matter which of the procedure’s return statements is being used.

## Entry Points

Entry-point instrumentation, which can only be “before” the entry point, executes after any before-procedure instrumentation, but before any other instrumentation in the procedure.

This instrumentation is associated with a specific entry point and only executes when that point enters the procedure. Each entry point represents a specific instruction, but the entry-point instrumentation won’t be executed unless that instruction is reached by a call from another procedure.

## Call Sites

Instrumentation added before a call site executes before the procedure is called, and thus before any instrumentation in the target procedure.

Instrumentation added after the call site executes upon the target’s return. Note that some procedures might not return—for example, they may execute a `longjmp()` to return to a previous context. Instrumentation added after a call site to one of those procedures will never be executed. You can check the site’s `atck_call_returns()` attribute to see whether the target is known to return.

Call sites sit in between two basic blocks: the procedure is called after the end of the first block, and then returns to the beginning of the next block. Instrumentation added before the call site executes after instrumentation added to the end of the basic block containing the site. Instrumentation added after the call site executes before instrumentation added to the beginning of the next basic block.

## Basic Blocks

This one, at least, is simple: Instrumentation added before a basic block executes before the first instruction in the block. Instrumentation added after a basic block executes after the last instruction in the block.

## Instructions

Hang on. Instructions are involved enough that they really deserve their own separate section...

## *Instrumentation Sequencing—Instructions*

...ah, that's better.

## Instructions

ATtaCK makes an instruction's instrumentation look as much as possible like a part of the instruction itself. Instrumentation added before an instruction is the very last thing to be executed before the instruction itself. Instrumentation added after an instruction will get executed immediately after the instruction, before anything else in the program.

This can get weird with instructions that change the flow of control, by branching or by calling a procedure. Instrumentation added *before* a jump is easy—the instrumentation executes, then the instruction, then the jump. But what does it mean to add instrumentation *after* a jump instruction?

To understand that, we will need to remember dynamic arguments. An instrumentation call can pass the contents of registers to the analysis routine. For instrumentation added after an instruction, any changes that instruction makes to registers are visible to the analysis routine.

For example, consider the following instruction:

```
xor a0, a0, a0
```

Let's assume that we have an analysis routine that expects to be passed the contents of r1. Instrumentation added before this instruction will pass in the original value of r1. Instrumentation added after the instruction will pass the new value, which in this case will be zero.

So instrumentation added after a jump has access to the new contents of all the registers, *including the program counter*. The instrumentation actually happens in between the change to the program counter and the transfer of execution. Likewise, for a call instruction such as jal, which stores the return address in r31, the instrumentation call can pass the new values of both the program counter and r31.

On many chips, including the EE, when a register is loaded from memory, the new value isn't visible for several cycles afterward. If an instrumentation call after an instruction passes the loaded register to an analysis routine, ATtaCK adds

in a delay so that the new value is available. This can cause a significant slow-down of your application, so only read registers when truly necessary.

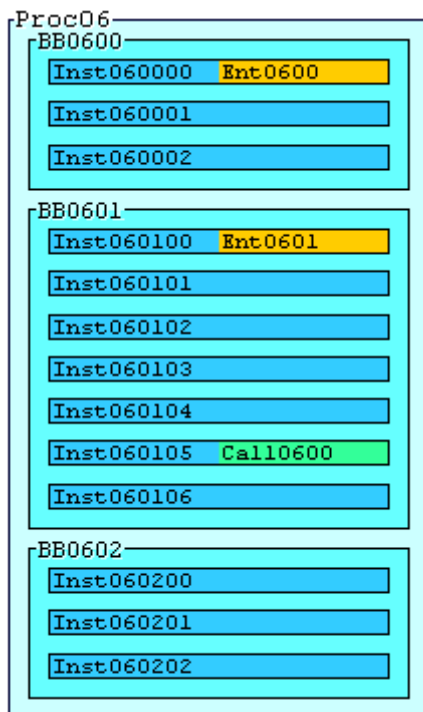
Similarly, most branches on the EE are *delayed*—the instruction after the branch executes while the processor prepares to transfer execution. You might think that instrumentation code added after a branch would displace your application’s code out of that “delay slot,” disrupting your code. It doesn’t. The instrumentation executes, then the delay-slot’s instruction (including any instrumentation *it* might have) executes, then the branch takes effect.

How does all this work? I have no idea. ATtaCK handles it all behind the scenes. The bottom line is that your tools can put instrumentation anywhere in the program you want, without worrying about what the actual CPU is doing.

Instrumentation might slow your program down, but it won’t *break* anything. If you forget how instrumentation sequencing works, your analysis code might not work the way you want. And if you write buggy analysis routines, they could break. But nothing you do in an instrumentation tool can crash your application.

## Pop Quiz

Here’s a quick test to review how all this works. Let’s diagram a simple procedure:



**Fig. 03-01: Diagram of a Simple Procedure**

Our instrumentation tool in this case is identifying objects using 32-bit values, somewhat like IP addresses. Here we see procedure 06, which contains basic block 0600, which in turn contains instructions 060000 through 060002, and so forth.

Let’s assume that every object in the diagram is instrumented both before and after, with a separate analysis routine for each level and position. Thus, the routines are

`BeforeProc()`, `AfterProc()`, `BeforeEnt()`, `BeforeBB()`, `AfterBB()`, `BeforeCall()`, `AfterCall()`, `BeforeInst()` and `AfterInst()`. Each routine takes the ID value of the instrumented object.

For the sake of discussion, call site 0600 calls procedure 1E, which is not itself instrumented below `BeforeProc()` and `AfterProc()`.

Now, if the caller enters this procedure at entry point 0600, what analysis routines will be called, and in what order? The answer in the Quiz Answers section of the document, but try to work this out on your own first.

## Creating Analysis Code

ATtaCK analysis code is a simple C program that contains routines, which will be invoked by your instrumentation calls. There are three steps to creating this program: First you set up a CodeWarrior for PlayStation 2 project to compile and link the program. Then you design the analysis code itself, writing standard C routines. Finally, you tell ATtaCK the names and argument lists of your analysis routines, so that it can set up the instrumentation calls properly.

## Building Analysis Code

Your analysis routines live in a small, temporary executable file, created using a separate CodeWarrior for PlayStation 2 project. Back in Lesson 01, you learned how to set up project stationery to create new analysis-code projects. That same project stationery should work for just about every analysis tool you write, so setting up the project and building the code is easy.

Now that you actually understand what instrumentation and analysis code are trying to do, you might want to go back and review Lesson 01. Pay particular attention to the project settings for analysis code, because they shed some light on how things work. For instance, the linker must be told to use the `atcktarg_start` library routine as the entry point for the analysis “program,” since there is no `main()` function in analysis code.

If you want to review Lesson 01 now, feel free—I’ll wait here. From this point forward, I’ll just assume you understand and are using the project stationery.

## Designing Analysis Code

The analysis code for a particular tool is a small C program that you write. This program does nothing by itself—it has no `main()`, and doesn’t even call any of its own routines. Instead, your instrumentation tool uses ATtaCK to attach the analysis code onto the end of the target application, inserting calls to that code throughout the program.

When instrumentation code calls an analysis routine, what does that routine do? The simplest analysis routines just increment counters. In fact, that’s pretty much what *all* analysis code does. You want your analysis routines to be as lightweight as possible. Simply incrementing a counter, leaving any real processing of the data for later, is usually the best approach.

More often, the analysis routine increments one element from an *array* of counters. That way, the instrumentation tool can use a numeric ID to specify which counter should be incremented. The alternatives are to use a separate function for each counter, which is cumbersome, or to identify a counter by some non-numeric means (such as a string), which is inefficient. Analysis routines should be easy to write and fast to run, and numerically indexed arrays fit the bill perfectly.

The `ProcCount()` function we saw incremented one element of `ProcCounts[]`, an array with one element per procedure. Each procedure was identified by a



numeric value that was passed into `ProcCount()`. These values came from the tool, and only had meaning to it. Neither the instrumentation calls nor the `ProcCount()` analysis routine understood the ID value to mean anything other than which counter in the array to increment.

This brings up an important point: The data gathered by the analysis code almost always is meaningless without the instrumentation tool. All of the context information—the table that translates counter ID numbers into procedure names, for example—lives there, not in the analysis code. After your instrumentation tool collects the results of the analysis code, it's your job to attach the context information to those results before saving or displaying them.

## Simple Analysis Code

Here's the simplest analysis code you're ever likely to use:

```
#include <atcktarget.h>
#pragma force_active on
atck_uint32_t ProcCounts[1000];
void CountProc(atck_uint32_t procID)
{
    ProcCounts[procID]++;
}
```

The first thing to notice about the program is that it is cross-platform: It doesn't use any types that aren't defined by ATtACK. You can get away with using standard C types in an instrumentation tool. The whole point of analysis code, on the other hand, is to gather data to communicate back to the host system, and it's vital that the basic types be the same on both sides of that connection.

In particular, instrumentation calls—which are essentially calls from your instrumentation tool to the analysis code, across the gap between the host and target—can only pass ATtACK's types. Analysis routines cannot use standard C types in their argument lists. We'll cover the range of available data types later on, but `atck_uint32_t` is the one you'll use most often anyway.

The ATtACK types and interfaces are defined in the system header `atcktarget.h`, included at the top of this program. Your instrumentation tools, which run on the host system, include `atck.h`; analysis code, which runs on the target, uses just a subset of the ATtACK interfaces, defined in `atcktarget.h`. Some PlayStation 2-specific definitions are included in `atcktargetps2.h`, which you might need to include in addition to (not instead of!) `atcktarget.h`. This program doesn't use any of those, and we won't encounter a program that does until Lesson 05.

The pragma directive on the next line tells the linker to include all the routines from this file, even though none of them ever get called. In the final executable, there will definitely be instrumentation calls that use these routines, but the linker doesn't know that right now.

The top two lines are boilerplate—every analysis program you create will use them. The remaining lines are the actual analysis code.

The program simply creates a static array of 1000 32-bit unsigned integer counters. Instrumentation calls to `CountProc()` specify which counter to increment, and `CountProc()` dutifully increments the specified counter. Note that, like all analysis routines, `CountProc()` doesn't return a value.

See, I told you it was simple.

## What's Wrong With This Picture?

Now, if it were *really* that simple, you wouldn't need this course. This code is in fact *too* minimal, and in later lessons we'll see how to improve it.

For one thing, allocating a static array is a bad approach—1000 elements will either be too many, which is inefficient, or too few, which is downright disastrous. So “real” analysis code needs to work with dynamically sized arrays. You can create the array yourself, or you can let ATtACK handle it for you.

Another problem with this function is that it only gathers one piece of information—which procedure needs to be counted. So the only profile we can get from this code is the number of times each procedure was called. Any advanced optimization will certainly require more detailed data than that. At a minimum, we'd want to know how many instructions within each procedure actually got executed.

What's *not* missing, on the other hand, is code to communicate the results back to the host system. The analysis routines don't talk to the host at all. Rather, the host reads the memory of the target application (including its attached analysis code) directly. In this example, after we've finished profiling the application, the instrumentation tool will look up the address of the `ProcCounts[]` array and read the counters out of it. That process is the subject of Lesson 04.

At any rate, while there's a lot more to learn about designing analysis code, we can get by with our simple `CountProc()` function until Lesson 05, when we cover analysis code in detail.

## Declaring Analysis Code

Analysis routines have to be declared twice: first in C, for the benefit of the compiler, and then in a C-like description language, for the benefit of ATtACK.

The C declaration is straightforward, as we saw in the `CountProc()` example above. Now when I say “declaration” here, I'm talking about “`void CountProc(attack_uint32_t procID)`”—the function's signature. If the body of the function itself follows that, then it's more precisely a “definition” rather than a “declaration.” Since analysis routines seldom call each other, you generally won't bother *declaring* the routines. Nevertheless, I'll continue to refer to this as the C declaration, to strike a parallel with the ATtACK declaration.

Your instrumentation tool *will* call the analysis routines—indirectly, through instrumentation calls added to the target application. So the routines need to be declared to the instrumentation tool. In this case, it's ATtACK, rather than the C compiler and linker, that is generating the call, and so you have to use ATtACK function signatures rather than C ones.

An ATtACK declaration is represented by, you guessed it, an object—the `atck_proto_t`, or “prototype” object. The prototype object contains all the information ATtACK needs to know about a particular analysis routine.

## Declaring ATtACK Prototypes

Prototype objects are constructed using a factory method of the program object, like so:

```
atck_proto_t* atck_analproto(atck_prog_t* self, const char*
declare)
```

As a method of the program object, this takes a handle to the program in question as its first argument. The second argument is a string containing the function declaration, written in a C-like definition language described below.

If the declaration is valid, this method returns a handle to the new prototype object. This handle is then passed in as the *targ* argument for an instrumentation call, as seen earlier. If the declaration is invalid, this method returns `NULL`, and ATtACK issues a diagnostic message to `stderr`. The most obvious reason why a declaration would be invalid is a syntax error.

Remember that when you opened the program you also specified at that time the name of the analysis program to use. If you didn't specify an analysis program, or if the declared function doesn't exist in that program, this method will return `NULL`.

Prototype handles are owned by the program object, and are released when it is. You do not need to release them yourself. In fact, you *can't* release them yourself. So there!

## Definition Language

ATtACK function declarations use a C-like language, like so:

```
void name(arg1, arg2,...)
```

Analysis routines never return anything, so the void is boilerplate. *name* is the function's name, exactly as it appears in the analysis program's symbol table—so if you write your analysis routines in C++, you'll have to mangle the name for ATtACK. (And the moral of that story is “Don't write your analysis routines in C++.”)

Each of the function's arguments are specified by type only—no names. The arguments must use ATtACK's basic types, not their standard C equivalents. Types are specified by their core name—the middle part, in between the “`atck_`” and the “`_t`.” For example, the extremely common `atck_uint32_t` appears as just `uint32` in function declarations.

All analysis-routine arguments must be 32-bit or 64-bit values. They fall into three categories: static arguments, which are values passed directly from the instrumentation tool to the analysis routine; dynamic arguments, which are values passed from the target application to the analysis routine; and arrays, which are pointers to memory buffers allocated within the target application's data space.

We'll stick with static arguments for now, since they're the ones you'll use most often anyway. Dynamic and array arguments will be covered in Lesson 05.

The static arguments available for analysis routines are listed in Table 03-01.

| ATtaCK Type                 | Prototype Token      |
|-----------------------------|----------------------|
| <code>atck_int32_t</code>   | <code>int32</code>   |
| <code>atck_int64_t</code>   | <code>int64</code>   |
| <code>atck_uint32_t</code>  | <code>uint32</code>  |
| <code>atck_uint64_t</code>  | <code>uint64</code>  |
| <code>atck_addr_t</code>    | <code>addr</code>    |
| <code>atck_float32_t</code> | <code>float32</code> |
| <code>atck_float64_t</code> | <code>float64</code> |

*Table 03-01: Valid Types for Analysis Routine Static Arguments*

## A Minimal ATtaCK Tool

The simplest possible profiler would just count the number of times any procedure in the application was called. Each procedure would be counted the same whether it was a three-line swap function or a 500-line AI state machine. We'd add an instrumentation call at the top of each procedure that invoked a very simple analysis routine:

```
void CountProc(void)
{
    NumProcsCalled++;
}
```

In fact, if you remember the example from Lesson 01, which is exactly what BareBones did. Not very useful. In the sample program for *this* lesson, we'll do a little better than that. The tool we're going to look at here counts the number of times *each* procedure gets called. We still add an instrumentation call at the top of each procedure, but we need that call to identify *which* procedure was entered. For efficiency, we'll identify procedures by a numeric ID, which gets passed to the `CountProc()` routine we've been looking at all lesson.

That routine uses the numeric ID to increment one counter out of an array. At the end of the run, we can read that array, map each numeric ID back to the original procedure name, and find out how many times any given procedure was called.

How useful a profiler is this? So-so. The more often an individual procedure is called, the more deserving of optimization it's likely to be. That's not universally true—procedures with loops being the obvious problem. It's good enough for a first cut, though.

## Creating the Project

Unlike Display from the previous lesson, this sample program wasn't included with the ATtaCK installation, so you'll need to create a new project for it yourself. This is done using the same process we saw in Lesson 01: Use the "ATtaCK Metaproject" stationery we set up then to create a new metaproject, instrumentation-tool project and analysis-code project. Name this project (and the .exe file it produces) ProcCount.

You'll need three files for ProcCount: `ProcCount.c`, `ProcCount.h` and `CountProc.c`. These can be downloaded from this course's supplemental-material folder. Delete `main.c` from your instrumentation-tool project and replace it with `ProcCount.c`. Then delete `analmain.c` from the analysis-code project and replace it with `CountProc.c`. Once the code is ready, select the metaproject (`ProcCount.mcp`) and make it—hit F7 or select `Project > Make`. Refer back to Lesson 01 for troubleshooting.

Once you're sure everything builds correctly, open up `ProcCount.c` and let's take a look!

## *ProcCount: Navigation and Instrumentation*

This is a simple program—everything is done in `main()`, and you've seen all of this code at least once already.

After the system and ATtaCK headers, the file includes `ProcCount.h`. Open that up and you'll see that it has just one line, defining the `MAXPROCS` constant. More on that in a minute. Let's go back to `ProcCount.c`.

At the top of `main()`, after all the variables are declared, we find the standard boilerplate—open a session, open a configuration file and open an application. The target application filename is read off the command line. The one thing that's changed is that, since we're now adding instrumentation calls, we need to tell ATtaCK where to find the analysis code—in this case, it's in the file `"<tooldir>..\Analysis Code\analcode.elf"`.

The filename `analcode.elf` is obviously the output executable file we set up in the analysis-code project. ATtaCK replaces the `<tooldir>` macro with the directory that the instrumentation tool is in. The `".."` steps up to the metaproject directory, and the `"\Analysis Code\"` steps back down into the analysis-code directory where `analcode.elf` is.

The first thing we do with the newly-opened program handle is find out how many instrumentable procedures it has. Each instrumented procedure will be instrumented with a call to our `CountProc()` routine, which you'll remember uses a static array of 1,000 elements. Actually, as you'll see when we look at `CountProc.c`, we've updated that a little bit—now it uses a static array of `MAXPROCS` elements. By using a header file to share this value between the two programs, we keep them in sync with each other.

If our tool added instrumentation calls to more procedures than the analysis code can handle, `CountProc()` would crash. So we need to make sure that the instrumentation tool knows and respects the analysis code's limits. If the target application has more than `MAXPROCS` instrumentable procedures, we print an error message and quit.

Next we declare our single analysis routine, `Pcount()`. It's a void function that takes a single 32-bit unsigned integer. The declaration gives us a prototype handle that we'll use later on.

Now we iterate over every image in the program, and over every procedure within each image. I can't imagine what's left to say about iteration after our last lesson, so let's move on...

At the heart of the iteration loop, we check to see whether the given procedure is instrumentable. (Procedures not compiled by CodeWarrior—most notably, the Sony libraries—are not.) If it is, then we use the procedure's "call before" instrumentation method, `atck_proc_callbefore()`, to insert a call to `Pcount()` at the start of the procedure.

The instrumentation method takes three arguments, a handle to the procedure, a handle to the prototype of the routine to call, and the argument to pass in to that routine. Here we pass in `iproc`, a counter, which we then increment. The net result is that for the first procedure we pass 0, for the next 1, then 2, 3 and so forth. `CountProc()` will use this value to identify the procedure.

As we finish with each image, we call `atck_img_write()`, which saves the new, instrumented image, rather than `atck_img_release()`, which would discard our instrumentation.

Once all the images are done, we call the program object's `atck_finish_write()` method to close the new executable file. This method returns a handle to the new file, which we will use to download and run that program.

Next we connect to the target system, download the instrumented application and start it running. While the program is running on the T10000, the instrumentation tool waits for the user to hit enter on the PC. Then the tool halts the target so that we can read the results. All of this code is essentially boilerplate, so we'll save it for Lesson 04.

After we've stopped the target, we find the address of its `ProcCounts[]` array and upload it into our tool—another topic covered in Lesson 04. We can then read and display this array, but let's look at the analysis code that created it first.

## ***ProcCount: Analysis and Output***

### **Analysis Code**

The analysis code lives in `CountProc.c`. The only system header it needs is `atcktarg.h`. It also includes the shared `ProcCount.h` header, to get the `MAXPROCS` constant. That constant is then used to allocate the `ProcCounts[]` global array.

As discussed, each element of `ProcCounts[]` is an unsigned 32-bit integer counter. When a particular procedure in the target application is entered, an instrumentation call passes the procedure's ID to `CountProc()`, which increments the appropriate counter from the array.

And now back to the instrumentation tool.

### **Output**

The `atck_readdata()` function reads the entire contents of `ProcCounts[]` from the target into a buffer on the host. Arguments to the function tell ATtaCK that the memory consists of an array of unsigned 32-bit integers, so if the host and target had different byte orders, ATtaCK would be able to byte-swap each array element automatically.

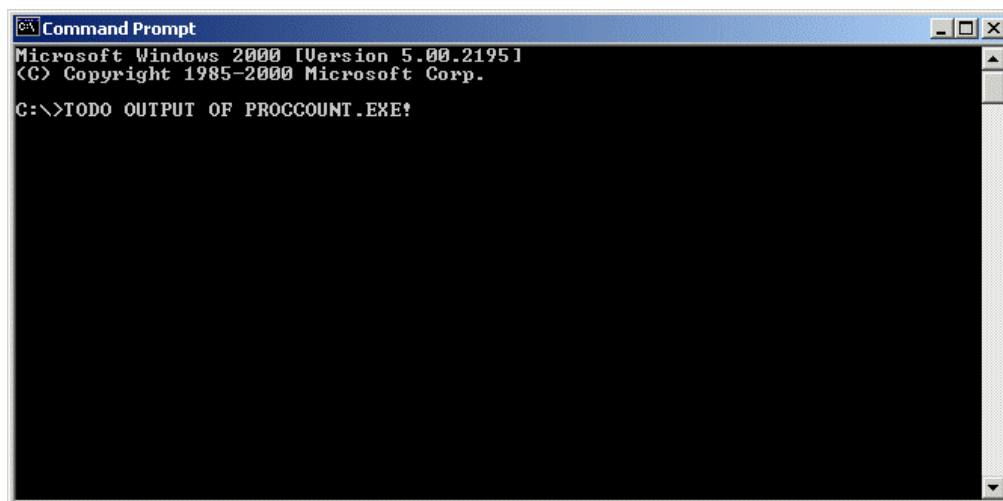
Of course, since we're talking to a PlayStation 2 from a PC, we don't need to swap the bytes. But it's a handy thing to remember for the future.

Once the buffer has been loaded, we iterate through the program again in order to attach a procedure name to each slot in the array. At first glance this seems weird and inefficient—why didn't we create an array of strings and store the procedure names during the first iteration pass?

To ask the question is to answer it: That would have been more work. The iteration is guaranteed to be the same each time through, so why *not* just do it twice? The big savings in programming time overwhelms the teensy tiny waste of computer time.

This highlights one of the most important ATtaCK principles: *Keep it simple, stupid!* It's not likely anyone is paying you to create elegant tools; they're paying you to analyze the application. Take the most direct route to that goal you can.

At any rate, having looked up the procedure name for each counter in the array, we simply print the two out. The result, shown in Fig. 03-02, is human-readable, but only really useful for small programs. A better output format would have been comma-separated values, which could then be loaded into a spreadsheet program and analyzed or at least sorted.



*Figure 03-02: ProcCount Output*

## Onward!

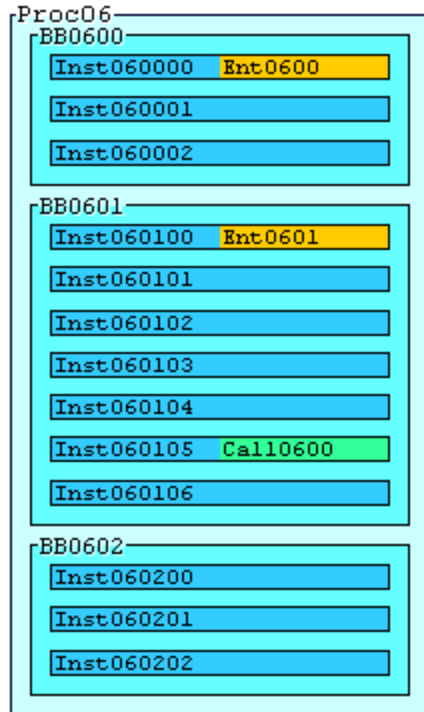
We're not quite halfway done, but we might as well be. Downloading and executing the target application is our next subject, and compared to navigation or iteration, it's trivial. Truth be told, it's so easy that I'm embarrassed to be getting paid for explaining it to you.

Not so embarrassed that I turned down the money, mind you. And since I *am* getting paid, let's go ahead and move on to Lesson 04: Running and Analyzing an Application.



## Lesson 03 Assignment

Here's a diagram of a simple procedure:



Our instrumentation tool in this case is identifying objects using 32-bit values, somewhat like IP addresses. Here we see procedure 06, which contains basic block 0600, which in turn contains instructions 060000 through 060002, and so forth.

Let's assume that every object in the diagram is instrumented both before and after, with a separate analysis routine for each level and position. Thus, the routines are `BeforeProc()`, `AfterProc()`, `BeforeEnt()`, `AfterEnt()`, `BeforeBB()`, `AfterBB()`, `BeforeCall()`, `AfterCall()`, `BeforeInst()` and `AfterInst()`. Each routine takes the ID value of the instrumented object.

For the sake of discussion, call site 0600 calls procedure 1E, which is not itself instrumented below `BeforeProc()` and

`AfterProc()`.

Now, if the caller enters this procedure at entry point 0600, what analysis routines will be called, in what order and with what arguments? The answer is below, but try to work it out for yourself first.

### Hint:

There are a few "gotchas" you need to watch out for:

- Entry point instrumentation only gets called if the procedure is actually entered there.
- A call instruction finishes before the call itself takes place.
- The instruction in a call's delay slot also happens before the call.
- A basic block ends when its last instruction ends, before any call out of the block takes place.

**Answer:**

```
BeforeProc(06)
BeforeEnt(0600)
BeforeBB(0600)
BeforeInst(060000)
AfterInst(060000)
BeforeInst(060001)
AfterInst(060001)
BeforeInst(060002)
AfterInst(060002)
AfterBB(0600)
BeforeBB(0601)
BeforeInst(060100)
AfterInst(060100)
BeforeInst(060101)
AfterInst(060101)
BeforeInst(060102)
AfterInst(060102)
BeforeInst(060103)
AfterInst(060103)
BeforeInst(060104)
AfterInst(060104)
BeforeInst(060105)
AfterInst(060105)
BeforeInst(060106)
AfterInst(060106)
AfterBB(0601)
BeforeCall(0600)
BeforeProc(1E)
AfterProc(1E)
AfterCall(0600)
BeforeBB(0602)
BeforeInst(060200)
AfterInst(060200)
BeforeInst(060201)
AfterInst(060201)
BeforeInst(060202)
AfterInst(060202)
AfterBB(0602)
AfterProc(06)
```

## Lesson 03 Quiz

1. Which of the following is an advantage that sampling has over instrumentation?
  - A. Speed.
  - B. Convenience
  - C. Accuracy
  - D. Selectivity
2. True or false: Instrumentation can be added both before and after entry points.
  - A. True
  - B. False
3. True or false: Calls added to the same object execute in the order they were added.
  - A. True
  - B. False
4. True or false: Instrumentation added after a call site is guaranteed to be executed.
  - A. True
  - B. False
5. Which of the following lines of code is *not* required in ATtaCK analysis code?
  - A. `#include <atcktarget.h>`
  - B. `#include <atcktargetps2.h>`
  - C. `#pragma force_active on`
  - D. None of the above
6. True or false: Analysis routines are unable to communicate with the host directly.
  - A. True
  - B. False
7. Which of the following is a valid return type for an analysis routine?
  - A. `int`.
  - B. `atck_uint32_t`.
  - C. `void`.
  - D. `void*`
8. True or false: You have to be careful when instrumenting branches on the PlayStation 2, to avoid displacing the instruction in the “branch delay” slot.
  - A. True
  - B. False

## Lesson 04: Running and Analyzing an Application

---

The ATtaCK framework handles all the details of downloading and running your instrumented application on the T10000. In our fourth lesson, you'll learn how this works, and how to pause, resume and kill a running application. We'll also look at how to communicate data between your PC and the target, and how to display your profiling results in a useful form. Finally, we'll explore SimpProf, a simple profiler.

### *Execution Concepts*

#### Lesson Objectives

Okay, wake up! This lesson is going to go by so fast that if you nap, you'll miss it. We're going to cover the penultimate two steps of the ATtaCK process: downloading and running applications, and reading data from the target system. For both of these, ATtaCK does almost all the work, so all you need to learn is a handful of API calls.

#### Download

The end result of instrumenting the target application is a new program, which will need to be run on the target system to perform the analysis. You can just launch it as you would any other program, but if you did that, you wouldn't be able to read your data. The analysis code has no way of saving data or sending it to the host system, nor can it communicate with the main application.

The only way to get the analysis results out of the program is to read them directly from an ATtaCK tool. And to do that, you have to have downloaded and executed the application from that tool in the first place.

You don't have to download and run the program immediately after instrumenting it, however. You could create a tool that ran an already-instrumented application. This would let you instrument the application once, save it, and then run it multiple times—even run it on multiple systems.

If you were analyzing AI routines, for instance, you might want to have several different play testers run the program, trying out different tactics. The aggregate results from all their sessions would tell you more about your AI code than the results from any single session.

As we've already seen, though, the tool that handles the execution needs to know exactly what the analysis code is like in order to know what memory buffers to read. So in practice, you usually download and run the application from the same tool that performed the instrumentation. Avoiding the nominal inefficiency of re-instrumenting the application every time the tool runs isn't worth wasting a lot of programming effort.

ATtaCK handles all the details of communicating with the target system. Your tool simply opens a connection to the target and specifies the application to run. ATtaCK downloads the application and immediately pauses it.

## Execution

Once the application is on the target, you can instruct ATtaCK to start execution. You don't have much fine control over the target, but you do have absolute power—you can tell it to start and stop, and your commands will be obeyed.

You should run the application for a little while in order to build up a body of data. You could start and stop the program based on a timer, but the better approach for a game is to run the program until the user halts it. You'll generally want to analyze "typical" gameplay, and the user is the best judge of when he's accomplished that.

When the target stops, whether the application exited, the operating system killed it or ATtaCK halted it, the tool gets sent an event. It then responds to that event by reading the analysis results from the target system.

## Reading Data

ATtaCK doesn't know in advance where in the target system's memory the application will wind up; it gets that information back from the target when the application is downloaded. An API function lets you look up the target address of a particular symbol.

Once you have an address, you can read data from the target. We saw this in the sample program last lesson, and there's really not much more to know. In addition to reading from the target, ATtaCK can also write to the target.

## Displaying Data

Analysis code needs some kind of context information to work properly. In the ProcCount sample program, the procedure-counting routine needed to know *which* procedure to count. To keep the analysis code as streamlined as possible, this context information is usually just a simple integer, which gets passed from the instrumentation tool to the analysis code. The tool is responsible for mapping these ID numbers back to actual code structures once it reads the data.

The easiest way to do that is simply to iterate through the code again, in the exact same sequence the code was instrumented. Most ATtaCK tools thus have two nearly-identical iteration loops: One runs through the application's code objects to add instrumentation calls, the other runs through to map analysis results back to the code objects.

Once the analysis results are associated with the original code, the tool can either dump this information to the display or a file, or it can perform higher analysis. A profiler that gathered instruction-counts for each procedure, for instance, might then go on to calculate the percentage of total execution time happened in each

procedure. It might even create totals and subtotals, to show how many cycles were consumed by each procedure plus its children.

## Executing an Application

To execute an application, you must first open a connection to the target system. The “device connection” object, `atck_dev_t`, represents the connection. This object is created using a factory method of the session object, like so:

```
atck_dev_t* atck_connect(atck_sesn_t* self, atck_cfg_t*
    config,
                        void
    (*evthandler)(atck_tvt_t*,void*,void*),
                        void* devID)
```

The first argument is a handle to the session object, as befits a session method.

The second argument is a handle to a configuration object, used to wrap the various system settings used by ATtaCK, such as the IP address of the T10000. Configuration objects are discussed in detail next page.

The third argument, *evthandler*, is a pointer to an event-handler callback function. We’re going to cover events in just a bit, so just take this as a given for now. You’re also free to leave this as `NULL`, and indeed many programs do just that.

The fourth argument is a pointer to an arbitrary variable. This variable is a user-specified (by which I mean *you*-specified) device ID. If you have multiple device connections but only one callback function, this device ID can be used to identify the connection for a specific event. This only matters for the event handler, so again, we’ll save it for later. If you’re not using an event handler, you may as well leave this argument `NULL`, too.

If ATtaCK can’t open a connection to the target system—if, for example, the target system is turned off—this method returns `NULL` and prints a diagnostic message. Otherwise, it returns a handle to the new device connection.

## Downloading the Program

The connection object is used to download a program to the target, using the following methods:

```
atck_run_t* atck_idownload(atck_dev_t* self, atck_iprog_t*
    prog,
                        atck_cfg_t* config, void* progID)

atck_run_t* atck_udownload(atck_dev_t* self, atck_iprog_t*
    prog,
                        atck_cfg_t* config, void* progID)
```

These functions are methods of the device-connection object, and thus take a connection handle as their first argument.

The second argument is a handle to an instrumented-program object; more on that in a moment. The `atck_idownload()` method expects the program to include instrumentation symbols and other special ATtaCK-created information; the

`atck_undownload()` method can be used with any program, whether it was instrumented by ATtACK or not.

The third argument is a handle to a configuration object, containing options for use with the downloaded program. On the PlayStation 2, those options are the host directory to load from, and the command line to pass to the downloaded application. This is covered in the next section.

The fourth argument is a pointer to a user-specified variable. Like the *devID* argument of `atck_connect()`, this pointer is passed to the event handler, where it can be used to identify which downloaded program generated a particular event.

If this method succeeds, it downloads the program to the target system and returns a “running program” object, `atck_run_t`.

If the method fails, it returns `NULL` and prints a diagnostic message.

## Opening Instrumented Programs

The “instrumented program” object, `atck_iprog_t`, represents a downloadable program. You may remember that the `atck_finish_write()` method returns a handle to the instrumented program it writes out. You can also open an existing program—instrumented or not—for downloading like so:

```
atck_iprog_t* atck_iopen(atck_sesn_t* self, const char*
filename,
                        atck_cfg_t* config)
```

This function is a method of the session object, and takes a handle to that object as its first argument. The second argument is the filename to open. The third argument is a configuration object, but since like `atck_open()` this method doesn’t accept any options, you should just pass `NULL`.

If the method succeeds, it returns an instrumented-program handle for use with the download functions. Otherwise, it returns `NULL` and prints a diagnostic message.

## Running the Program

As mentioned, the run object represents a downloaded program on the target system. The program begins already halted; to start it running, you’d call the `atck_continue()` method of the run object. That function, and the run object in general, is covered a little later.

Your tool’s execution continues while the program runs on the target system. Usually, you’ll want your tool to sit in an idle loop. The best way to do this is to call a blocking character-input library routine such as `getch()`; when the user hits ENTER, the function returns, at which point your tool would halt the target system.

## Ending the Run

While the program is running, you can start and stop it. When you've stopped it for the final time, you can shut it down permanently and release the run object by calling that object's "destructor," `atck_kill()`, as follows:

```
atck_bool_t atck_kill(atck_run_t* self)
```

If an error occurs, the method returns `ATCK_FALSE` and prints a diagnostic message; otherwise it returns `ATCK_TRUE`.

Finally, when you've finished with a connection object, you call *its* "destructor," `atck_disconnect()`:

```
Atck_bool_t atck_disconnect(atck_dev_t* self)
```

This method will call `atck_kill()` automatically for any programs still running. Like `atck_kill()`, it returns `ATCK_FALSE` and prints a diagnostic message if an error occurs.

## Configuration Objects

When working with applications and target systems, ATtaCK needs to know a number of configuration options—things like the host directory, the target system's IP address and so forth. These options are all platform-specific, so to preserve its cross-platform compatibility, ATtaCK encapsulates all the options into a single object, `atck_cfg_t`.

Configuration options can be set two ways. Most commonly, you pass a filename to the configuration object's "constructor," which parses that file for options. You can also use methods to add new options to an existing configuration object, allowing you to use the command-line or a different file format rather than the default ATtaCK format.

The configuration object also has methods to let you read its contents. Thus, you can use these objects to store tool settings and other non-ATtaCK options.

## Configuration Objects

### Initialization

To create a configuration object, you use a factory method of the session object:

```
atck_cfg_t* atck_config_new(atck_sesn_t* self, const char*
filename)
```

The first argument is the usual handle to the session object. The second argument is the name of a file from which to initialize the configuration object. To create an empty configuration object, pass `NULL` for the filename.

The filename argument can use absolute or relative paths, using the standard Windows file syntax. It can also use special keywords to identify known directories:



- The keyword `<syscfg>` is replaced with the full path and filename of the system configuration file—usually `lib\ps2\syscfg.txt` in your ATtaCK installation, as we saw back in Lesson 01. This is the file you should almost always load and pass to `atck_connect()`.
- The keyword `<tooldir>` is replaced with the path to the directory containing the tool’s instrumentation code.

The format of the configuration file and the options each ATtaCK function expects are described at the end of this section.

At the end of your program, you can free any configuration objects you created by calling their “destructor”:

```
void atck_config_free(atck_cfg_t* self)
```

However, this isn’t necessary—releasing the session object with `atck_endsession()` will automatically free any configuration objects created in that session.

## Reading Options

Configuration options are identified by name and come in two types, integers and strings. To read an option from a configuration object, you use the following methods:

```
long atck_config_getint(atck_cfg_t* self, const char* opt)
const char* atck_config_getstr(atck_cfg_t* self, const char* opt)
```

If the specified option doesn’t exist, or if it exists but is of the wrong type, these methods return 0 or `NULL`, respectively. The string returned by `atck_config_getstr()` is valid until the configuration object is destroyed, or until the specified option is changed to a different value.

To find out whether a particular named option exists, and/or to learn its type, use the following method:

```
atck_cfgtyp_t atck_config_hasopt(atck_cfg_t* self, const char* opt)
```

This method returns an enumerated type with three possible values: `ATCK_CFGTYP_INT`, if the option exists and is an integer; `ATCK_CFGTYP_STRING`, if it exists and is a string; or `ATCK_CFGTYP_NONE`, if the option doesn’t exist.

## Writing Options

If you’re using command-line options or your own configuration files, you can store these options in a configuration object by hand, using the following methods:

```
void atck_config_addint(atck_cfg_t* self, const char* opt, long val)
void atck_config_addstr(atck_cfg_t* self, const char* opt, const char* val)
```

The *opt* argument specifies the option's name, and *val* specifies its value. If an option with that name already exists, it is overwritten with the new value and type.

## Configuration Files

ATtaCK configuration files are plain text, much like Windows .INI files. Each line in the file defines an option and its value, like so:

```
ip = 192.168.0.10
addr = 0xfa001234
output = file.txt
message = " Analysis complete. "
prompt = "Hit "ENTER" to continue."
```

Each line consists of the option name, an equal sign and the option value; spaces and tabs within the line are ignored. If an option value is a recognizable decimal, hexadecimal or octal value, it is stored as an integer, otherwise it is stored as a string. Dotted-decimals, such as IP addresses, are interpreted as strings.

You can force an option to be interpreted as a string by surrounding it in quotes, but this isn't necessary for most strings. Putting a value in quotes will allow you to include leading and trailing whitespace, which would otherwise get stripped. The quotes themselves do not become part of the value, of course, although any quotes inside the string are preserved.

## Connection Options

The options expected by `atck_connect()` are the ones contained in `syscfg.txt`. You usually only need to modify this file when you install ATtaCK on a new machine, so read the installation instructions for more details. Here's a quick review:

`ip` (string): The IP address of the T10000 target system.

`port` (integer): The port number of the T10000

`priority` (integer): The connection priority of the T10000

`timeout` (integer): The connection timeout value, in milliseconds

`hostioDrive` (string): A single letter identifying the default drive for host files

## Download Options

The download functions `atck_idownload()` and `atck_udownload()` only expect two options, both of which are, well, optional:

`hostio` (string): The path to the application's host I/O folder. The default value for this option depends on how the downloaded program was opened. For existing programs opened with `atck_iopen()`, this defaults to the opened application; for programs "opened" as the result of `atck_finish_write()`, this value defaults to the location of the original, uninstrumented application.

`cmdline` (string): The command line arguments to pass to the application. This works just like an actual command line: Each argument is delimited with spaces, and you don't need to surround the line in quotes. The default for this option is simply a blank line.

## ATtaCK Events

### Event Handlers

To save you the trouble of sitting in a loop polling the status of the target system, ATtaCK provides an event system. The target system reports various events back to ATtaCK, which passes them on to an event-handler callback function you specify in your call to `atck_connect()`.

All event handler functions follow this pattern:

```
void MyEventHandler(atck_tevt_t* event, void* devID, void*
progID)
{
    /* Handle the event */
}
```

The function takes three arguments, a handle to the event generated, and the IDs of the device connection and program that generated that event.

There can be at most one event handler for any given connection—"at most" because you don't have to have an event handler if you don't want one. Conversely, if you had multiple connections, they could all share the same event handler. The device and program IDs let you tell which device or program generated which event.

These IDs are the values you passed in to the connect and download methods we saw previously. What they are is up to you. They could be pointers to integers, pointers to character strings, even pointers to C++ objects—ATtaCK doesn't care. Whatever value you pass in just gets passed back to you.

They could even be `NULL`. If you only intend to have one connection open and have one downloaded program running at the same time—most likely the case—then you can decide not to bother with this ID nonsense and can just pass in `NULL`s. Also, if the generated event isn't associated with a program, then the *progID* value would be `NULL` regardless of what you passed in to the download method.

In this lesson's sample program `SimpProf`, we'll see a clever use for the *devID* pointer. `SimpProf` stores the running application's context information—handles to the session, the uninstrumented program and the address of the data buffer—in a structure, and then passes a pointer to that structure through *devID*. When the event handler receives that pointer, it gets access to that information, which it uses to read, format and print the profile data.

## Designing Event Handlers

An event handler can more or less do anything you want. There are only a few restrictions.

First, the event object doesn't belong to you, so don't try to release it.

Second, the event handler function may—almost certainly *will*—be called from a different thread than your instrumentation tool. Take all the usual precautions in working with global data.

Last but not least, an event handler cannot call functions that generate events themselves! In practice, this means you can't call functions that stop the running program—`atck_disconnect()`, `atck_stop()` or `atck_wait()`—nor can you call functions that download a new program—`atck_idownload()` and `atck_udownload()`.

## Events

An event object, `atck_tevt_t`, which is passed into your event handler, represents the event. The first thing you should do is get the event type, as follows:

```
atck_tevt_typ_t atck_tevt_type(atck_tevt_t* self)
```

This is, of course, a method of the event object. It will return one of three values:

- **ATCK\_T EVT\_LOADPCT:** Events of this type indicate the progress of a downloading program, so that you can display a progress bar on the host. The percentage completion is available from `atck_tevt_loadpct()`, a method of the event object which returns a value from zero to 100. ATtaCK will try to send `LOADPCT` events as often as it can, but it only *guarantees* that you'll receive one: When the download finishes, you'll always get one of these events with a value of 100.
- **ATCK\_T EVT\_PRINT:** Your ATtaCK tool can print a diagnostic message whenever it feels one is called for. The ATtaCK code on the target may also decide to issue a diagnostic message. When that happens, it raises an event of this type. The message itself is available from `atck_tevt_printstr()`, a method of the event object which returns a constant character string containing the message to print. Your handler should respond to this event by immediately displaying this string to the user, generally via `stdout` or `stderr`. The strings returned by `atck_tevt_printstr()` are only valid until the handler returns. If you want to save these messages, copy them to an allocated buffer in your tool.
- **ATCK\_T EVT\_STOPPED:** ATtaCK sends you this event whenever the target application stops or terminates. This might be because the application crashed, because the application's analysis code halted itself, or because your tool halted the program.

It's possible that future versions of ATtaCK will generate more events. Your event handler should simply ignore any event type it does not recognize.

The event object has two more attributes beyond those mentioned above:

```
atck_dev_t* atck_tevt_dev(atck_tevt_t* self)
atck_run_t* atck_tevt_run(atck_tevt_t* self)
```

These methods respectively return the device-connection and running-program object that generated the event.

## ***Controlling the Application***

### **The Running-Program Object**

#### **Methods**

ATtaCK gives you only the most basic control over the target system: You can pause and resume the running program, and you can kill the program altogether. In particular, you can't perform the kind of line-by-line execution that you can with a debugger.

Within those limits, here's what you can do. All of these functions are methods of the running-program object, `atck_run_t`.

```
atck_bool_t atck_continue(atck_run_t* self)
```

This method continues execution of the program if it's currently stopped, and has no effect if the program is running. It returns `ATCK_FALSE` if the program cannot resume—for instance, because it's been terminated by `atck_kill()`—or `ATCK_TRUE` otherwise.

```
atck_bool_t atck_stop(atck_run_t* self)
```

This method halts the running program. If the program is in the middle of a critical section, as discussed on the next page, then this method will wait until the critical section ends before stopping the program. Notice that this means that, if the program *never* ends the critical section, this method will never return, and your ATtaCK tool will hang.

When the program finally does stop, your event handler will receive an `ATCK_T EVT_STOPPED` event. Thus, an event handler may not itself call `atck_stop()`.

Once the program has stopped, and after the generated event has been processed, this method will return `ATCK_TRUE`. If the program has already been killed or can't be stopped for some other reason, this method returns `ATCK_FALSE`.

```
atck_bool_t atck_wait(atck_run_t* self)
```

This method waits for the running program to halt itself, either by terminating normally, by crashing, or by calling the `atcktarg_stop()` analysis function (see notes). It is essentially a replacement for an idle loop—if your instrumentation tool has nothing to do until the application finishes running, you can call this method to wait for it to finish. By doing so, though, you give up the ability to halt the application from the host.

---

---

## Stopping the Application on the Target

The running-program object allows you to halt the application from the host side of the connection. There are two ways to halt the application on the target.

The first is for the application to end itself, the same way it normally would—by exiting `main()`, by calling `exit()`, by crashing—you name it. In this case, the program cannot resume, and calls to `atck_continue()` will fail.

The second is for an analysis routine in the target application to call `atcktarg_stop()`. This function simply halts the application and raises an `ATCK_TEVT_STOPPED` event on the host, and looks like this:

```
void atcktarg_stop(void)
```

A program stopped in this manner may be resumed from the host using `atck_continue()`. For obvious reasons, there is no `atcktarg_continue()` function.

One particularly useful technique is to create a “do-nothing” function in your PlayStation 2 application that gets called in response to a certain keypad button. In your analysis code, create a simple routine that does nothing but call `atcktarg_stop()`. Then insert an instrumentation call to that routine at the start of the do-nothing function. Pressing the keypad button now instructs ATtaCK to pause your application.

---

---

When the program finally does stop, your event handler will receive an `ATCK_TEVT_STOPPED` event. Thus, an event handler may not itself call `atck_wait()`.

Once the program has stopped, and after the generated event has been processed, this method will return `ATCK_TRUE`. If the program has already been killed, this method returns `ATCK_FALSE`.

```
atck_bool_t atck_kill(atck_run_t* self)
```

This method unconditionally halts the running program, without honoring any critical sections the program may have claimed. The program immediately stops, generating an `ATCK_TEVT_STOPPED` event. Thus, an event handler may not itself call `atck_kill()`.

This method also destroys the running-program object and frees up all resources associated with it. That means that you will not be able to read data from or resume the program after it has been killed.

You will usually call this method on an already-stopped program, after you’ve read the analysis data from it. If the program has already been killed, this method returns `ATCK_FALSE` otherwise it returns `ATCK_TRUE`.

## Attributes

...or rather *attribute*. The running-program object has one attribute, `atck_status()`:

```
atck_status_t atck_status(atck_run_t* self)
```

This access method returns the program's current status. You can call this inside an event handler, or to poll from within an idle loop. The return value will be one of these constants:

- **ATCK\_STATUS\_RUNNING:** The program is running.
- **ATCK\_STATUS\_TSTOPPED:** The program has been stopped by analysis code on the target, using the `atcktarg_stop()` function, and may be resumed with `atck_continue()`.
- **ATCK\_STATUS\_HSTOPPED:** The program has been stopped by the host, using the `atck_stop()` method, and may be resumed with `atck_continue()`.
- **ATCK\_STATUS\_EXITED:** The program has ended normally and may not be resumed.
- **ATCK\_STATUS\_FAULTED:** The program has ended abnormally, through a hardware fault or some other disaster. Not only can the program not be resumed, but its memory cannot be read, and any cleanup analysis code inserted via `atck_callafter()` probably wasn't executed.
- **ATCK\_STATUS\_ERROR:** A communications error occurred.

## Critical Sections

ATtACK provides special lock variables called *mutexes* (a contraction of “mutual exclusion”). These let your analysis routines establish critical sections, stretches of code that cannot be interrupted by another thread or by the host.

It's possible that your target application is multithreaded. If the application is multithreaded, and both threads are instrumented, then the analysis code is also multithreaded. If two threads tried to increment the same counter at the same time, your profile data would be corrupted. A mutex can be used to keep that from happening.

Even in a single-threaded application, your analysis code might want to prevent any interference from the host. For example, if you decide that you need a minimum of a hundred frames of profile data, you could lock a mutex and not release it until the hundredth frame of data had been gathered. Likewise, the target application might have a task that shouldn't be interrupted by the host, such as writing to NVRAM or disk.

On the other hand, it's perfectly possible you won't ever need to use mutexes. However, it's better to have the knowledge and not need it than need the knowledge and not have it, so let's look at them anyway...

## Mutexes

To create a mutex, your analysis code must first declare a variable of the type `atcktarg_lock_t`. This scope of this variable should encompass the location where you want the critical section to begin and the place where it ends. For

example, if your critical section is entirely within a single analysis routine, its mutex can be a local variable. For a critical section that spans two or more functions, however, you need to declare a global mutex.

The mutex needs to be initialized before it can be used. This is done with the following function:

```
void atcktarg_initlock(atcktarg_lock_t* mutexptr)
```

All you need to do is pass the address of the mutex to this function, which is guaranteed to succeed. However, make sure that you call this function from single-threaded code! The best place to initialize mutexes is in an analysis function called from before the program: An instrumentation call inserted using `atck_callbefore()` is guaranteed to run in a single thread before anything else in the application.

## Locking and Unlocking

Once a mutex is initialized, you can use it to protect critical sections. Upon entering a stretch of code that you wish to protect, you request a lock on the mutex. Once you receive a lock on a particular mutex, no other thread can lock it, nor can the host system halt your application. When the critical section finishes, you then release the mutex.

```
void atcktarg_lock(atcktarg_lock_t* mutexptr)
```

This function locks the specified mutex, and does not return until the lock is granted. That means that if another thread has the mutex locked, this thread will wait until the lock is released. If the lock is never released, perhaps due to a bug in your code, then this function will never return, and the calling thread will hang.

All interrupts are disabled for a thread that has a mutex locked. If any thread in the application has a mutex locked, the host cannot halt the target with `atck_stop()`—the target cannot even halt itself with `atcktarg_stop()`. However, `atck_kill()` will always work.

```
void atcktarg_unlock(atcktarg_lock_t* mutexptr)
```

If the calling thread has the specified mutex locked, then the lock is released, and any other thread waiting on that lock resumes execution. If the calling thread doesn't already have the mutex locked, nothing happens—you cannot release another thread's mutex.

## Designing Critical Sections

Let's look at how to use mutexes to handle the three scenarios outlined at the start of this page. In the first situation, analysis code called from multiple threads needs to increment the same array of counters. To prevent two threads from writing at the same time, the array of counters is “wrapped” in a mutex, like so:

```
atcktarg_lock_t TheMutex;

void CountProc(atck_uint32_t procID)
{
```



```

    atcktarg_lock(&TheMutex);
    ProcCounts[procID]++;
    atcktarg_unlock(&TheMutex);
}

```

When CountProc() wants to modify ProcCounts[], it requests a lock on the mutex, releasing the lock when it's done. If another thread's analysis code already holds the lock, then the first thread will wait for it to be released. In this situation, you should make sure that your code releases the mutex as quickly as possible, so that the threads don't bog each other down.

In the second scenario, the analysis code wants to ensure that it collects 100 frames of data before being interrupted by the host. The code would lock a mutex when it gathered the first frame and release it when it gathered the 100<sup>th</sup> frame, like so:

```

atcktarg_lock_t TheMutex;
atck_uint32_t NumFrames = 0;
void AnalyzeFrame(...)
{
    if (NumFrames == 0)
        atcktarg_lock(&TheMutex);

    /* Frame analysis code goes here! */

    NumFrames++;
    if (NumFrames == 100)
        atcktarg_unlock(&TheMutex);
}

```

In the final scenario, the target application performs certain tasks that must not be interrupted by the host. This presents more of a challenge, since the target application isn't aware of the analysis code and can't use ATtaCK's mutexes directly. To get around this, we have to create "analysis" routines that do nothing but lock and unlock mutexes on the application's behalf. Then, we insert calls to those routines at the appropriate spots in the application. Here's how the routines would look:

```

atcktarg_lock_t TheMutex;
void LockATCKMutex(void)
{
    atcktarg_lock(&TheMutex);
}
void UnlockATCKMutex(void)
{

```

```

    atcktarg_unlock(&TheMutex);
}

```

If the target application’s critical code all lives in known functions, then you would simply add an instrumentation call to `LockATCKMutex()` at the top of that function, and a call to `UnlockATCKMutex()` at the bottom.

Alternately, if you’re programming the application as well as the tool, you could write new “proxy” routines in the application. These routines, perhaps called `BeginATCKCriticalSection()` and `EndATCKCriticalSection()`, would just be empty placeholders. (You might have to put a short bit of do-nothing code in them to keep the compiler from optimizing them out of existence!) Your instrumentation tool would then add a call to `LockATCKMutex()` at the start of `BeginATCKCriticalSection()`, and a call to `UnlockATCKMutex()` to the end of `EndATCKCriticalSection()`.

## Communicating with the Application

Just as ATtaCK handles all the details of downloading and running programs on the target system, it also handles the details of reading and writing data. There might be other ways to get information off the target system, but these functions are far and away the easiest to use.

## Finding Target Symbols

ATtaCK lets you read directly from the target system’s memory. To do that, though, you have to know the address of the buffer you want to read. To do *that*, you must look up the address in the application’s symbol table:

```

atck_addr_t atck_anal_symaddr(atck_iprog_t* self, const char*
    sym)

```

This method of the running-program object returns the address of a symbol in the application’s analysis code. For example, in the `ProcCount` tool from Lesson 03, we saw this function used to look up the address of the `ProcCount[]` array. The symbol must be in the analysis code—if the symbol is in the target application itself, or if it doesn’t exist at all, this function returns `NULL`.

At this point, you might be thinking, “No problem. I can look up addresses from the target application already, using `atck_img_symaddr()`.” Well, that’s right and wrong. Remember, ATtaCK is adding instrumentation code. To your tool, that code is invisible, but to the actual application, it’s anything but. When ATtaCK writes out the instrumented application, it has to insert that code into your application, and it might move code around (changing its addresses) in order to make room. However, the PlayStation 2 version doesn’t do that—since all MIPS instructions are the same size, ATtaCK doesn’t have to move code to insert instrumentation. So *on the PlayStation 2 only*, you can rely on addresses from the uninstrumented application matching those in the instrumented application.

Oh, and one other thing: Only read—and especially write!—memory that you know to contain data, not code. Trying to read or write executable code may produce undefined results.

## Reading and Writing Data

Armed with the address of a buffer on the target, you're now ready to read data!

```
atck_bool_t atck_readdata(atck_run_t* self, atck_addr_t
    targaddr,
                                void* hostaddr, const char* type,
                                unsigned count)
```

This is a method of the running-program object, and takes a handle to that object as its first argument. The second argument is the source buffer to read from on the target, and the third argument is the destination buffer to copy into on the host.

The fourth and fifth arguments define the buffer's contents. ATtaCK expects you to be reading arrays. Rather than request a set number of bytes, you tell ATtaCK what type of data is in the array and how many elements (not bytes!) to copy.

The type string specifies the data type, using the same definition language used to declare prototype functions—"uint32" for an array of `atck_uint32_t`, "float64" for an array of `atck_float64_t`, and so forth. The array can also be composed of user-defined types (that is, `structs`—we'll cover these in Lesson 05).

If you want to copy raw bytes of data, of course, you can: Just specify "char" as the type and the number of bytes as the number of elements.

If the host and target have different byte orders, ATtaCK swaps the array elements appropriately as they're copied over. Between the PC and the PlayStation 2 you don't have to worry about this, but it's nice to know that you won't *ever* have to worry about it.

If ATtaCK can't read the target memory, perhaps because the address is bad or the connection has died, this function returns `ATCK_FALSE` and prints a diagnostic message. Otherwise it returns `ATCK_TRUE`.

```
atck_bool_t atck_writedata(atck_run_t* self, atck_addr_t
    targaddr,
                                void* hostaddr, const char* type,
                                unsigned count)
```

This method writes to the target. Other than the direction the data's going, it is identical to `atck_readdata()`. In fact, the argument list is exactly identical, which may throw some people off—if the ATtaCK API followed the C standard-library convention of "dest, src," the order of *targaddr* and *hostaddr* would be reversed for `atck_readdata()`.

## Arrays and Structures

Unless your analysis task is extremely simple, you'll be generating more than one variable's worth of data. Even the simple profiler from Lesson 03 used a

thousand-element array, and for some target applications that wouldn't be big enough. Allocating and managing data buffers is a vital part of creating an analysis tool. ATtACK provides two mechanisms to help manage your data: tool-allocated arrays, and user-defined data structures.

## Arrays

Since ATtACK's designers knew you'd be spending a lot of time dealing with arrays on both the host and target, they created an elegant system to allow you to "pass" arrays into your analysis code.

Basically what happens is this: Your instrumentation tool allocates an array in its memory however it sees fit—statically or dynamically. It's most efficient to allocate it dynamically, of course, but you can afford to be profligate with host memory.

This array is then passed to the analysis code as an argument in an instrumentation call. When that instrumentation call is added to the application, ATtACK allocates a buffer in the analysis code's data space the right size to hold the array, and copies the contents of the tool's array into it. The analysis routine then receives a pointer to that buffer. Usually this happens in an initialization routine, which then saves that pointer for the rest of the routines to use, but you can also pass read-only or throwaway arrays, such as strings.

## Declaring Arrays

To be able to pass an array to an analysis routine, you must declare it in the ATtACK prototype. Let's say that you have an array of unsigned 32-bit integers that you want to pass to an initialization routine. The prototype declaration for that routine would look like this:

```
pproto = atck_analproto(pprog, "void Init(uint32[])");
```

That's all it takes—just add the array marker to one of the existing types. In fact, you have a wider range of types available to you for use in arrays. ATtACK can only pass 32-bit or 64-bit values to analysis routines, but arrays, which are passed as pointers, get around this limitation. Table 04-01 shows the data types you can use with these arrays.

| ATtaCK Type                   | Prototype Token        |
|-------------------------------|------------------------|
| <code>char[]</code>           | <code>char[]</code>    |
| <code>atck_int16_t[]</code>   | <code>int16[]</code>   |
| <code>atck_int32_t[]</code>   | <code>int32[]</code>   |
| <code>atck_int64_t[]</code>   | <code>int64[]</code>   |
| <code>atck_uint16_t[]</code>  | <code>uint16[]</code>  |
| <code>atck_uint32_t[]</code>  | <code>uint32[]</code>  |
| <code>atck_uint64_t[]</code>  | <code>uint64[]</code>  |
| <code>atck_addr_t[]</code>    | <code>addr[]</code>    |
| <code>atck_float32_t[]</code> | <code>float32[]</code> |
| <code>atck_float64_t[]</code> | <code>float64[]</code> |
| <code>char*</code>            | <code>char*</code>     |

*Table 04-01: Array Types*

Two keywords can be placed in front of the array declaration to modify it. You can declare an array argument as being read-only with `const`; normally the analysis routine can write to the buffer it receives. You can also declare the buffer as `nopersist`, meaning that it can only be relied on for the duration of that one call to the analysis routine; the default is that the buffer is valid for the lifetime of the application.

## Using Arrays

There's a trick to passing an array into an instrumentation call. Normally, each argument passed to the analysis code corresponds to one argument to the instrumentation method. With an array, though, you first pass in the number of elements in the array, followed by a pointer to the array itself.

```
atck_uint32_t* ProcCounts = atck_calloc(psesn,
                                     atck_nproc_instr(pprog),
                                     sizeof(atck_uint32_t));

atck_callbefore(pprog, pproto, atck_nproc_instr(pprog),
ProcCounts);
```

This code fragment allocates an array of unsigned 32-bit integers, one for every instrumentable procedure in the target application. It then inserts a call to the `Init()` analysis routine into the start of the program. The one argument in the `Init()` routine's declaration has become two arguments in the instrumentation call: the number of elements in the array, and a pointer to the array.

The analysis routine, on the other hand, receives the array as a simple pointer of the appropriate type. ATtaCK does not automatically pass the number of elements to the routine. If you need that information, you can pass it yourself by declaring an additional integer argument. However, as you'll see in our sample program, you won't usually need the size of the array—by the nature of the iteration and instrumentation process, your analysis code won't be able to violate the array bounds.

When the analysis routine receives the array, it contains a copy of the array passed to the instrumentation call. Thus, you can use arrays to pass data *into* the analysis routines, in addition to storing data within the routines. Note, though, that passing the same array multiple times results in multiple arrays on the target—be careful not to use up too much memory! You can use the `const` keyword in the declaration to tell ATtaCK to collapse duplicate arrays into one.

Strings (indicated by `char*`) are a special type of dynamic buffer. Normally they function just like a `char` array. However, rather than having to specify the length of the array, ATtaCK assumes it to be null-terminated and calculates the length itself. Thus, strings are both passed and received as a single argument of type `char*`.

## Reading Arrays from the Host

Reading the contents of these arrays later is something of a challenge. When you allocate a static array in your analysis code, it gets a symbol you can look up; these tool-allocated arrays don't. Instead, the analysis code has to store the address in some location where the tool can find it. It does this by declaring a global pointer variable to store the buffer's address. When it receives the buffer, it stores the address in this pointer.

Knowing the name of this pointer, the instrumentation tool can use `atck_anal_symaddr()` to look up the pointer's address. It then uses `atck_readdata()` to read this one-element “array” of type `atck_addr_t`. *Now* it has the address of the actual buffer, which it can read normally.

## Structures

It's often easiest to manage data in structures. For instance, this lesson's example program, `SimpProf`, gathers for each procedure the number of times it is called and the number of instructions executed within it. This data could be stored in two parallel arrays, but the better approach is to store it in one array of structures, each of which contains the call count and instruction count for a single procedure.

ATtaCK lets you declare structures that you can then pass in arrays to your analysis routines. While you can *only* pass structures in arrays, there's nothing to stop you from passing an “array” of one element.

Much as with functions, structures must be declared in C to your compiler, and in a C-like syntax to ATtaCK. This is done with the following method of the session object:

```
atk_bool_t atk_analyse(atk_sesn_t* self,
                      const char* declaration)
```

The declaration syntax is a streamlined version of C: the structure name, followed by the types (but not the names!) of its members in braces. For example, a structure called `CallLog`, containing an address and an unsigned 32-bit integer, would be declared as `CallLog{addr, uint32}`.

The structure name must be a valid C identifier, with the additional restriction that it must also begin with a capital letter. Each member may be either one of the basic ATtACK types or another structure declared previously.

If the declaration is valid, the function returns `ATCK_TRUE`; otherwise it returns `ATCK_FALSE`. Unlike functions, there's no handle you need to save for later use—in effect, the name of the structure *is* the handle.

Note that it's extremely important for the C declaration of your structure to be the same on both the host and the target. To ensure this, you should put all your tool's ATtACK structures in a single header file, which is then included by both the instrumentation code and the analysis code. You should also define the ATtACK declaration string for each structure as a constant in the same file, to make sure the ATtACK declaration matches the C implementation.

---

## Memory Allocation with ATtACK

ATtACK includes a set of memory allocation methods to provide the same functionality as the standard C memory functions:

```
void* atk_malloc(atk_sesn_t* self, size_t bytes)
void* atk_calloc(atk_sesn_t* self, size_t elementsize,
                size_t elementcount)
void* atk_realloc(atk_sesn_t* self, void* mem, size_t
bytes)
void atk_free(atk_sesn_t* self, void* mem)
```

Each of these works the same as the C function of the same name (minus the “atk\_” prefix, of course). The only difference is that the allocation routines never return `NULL`. If they fail, the tool terminates and ATtACK prints a diagnostic message. This saves you the trouble of checking the return value against `NULL`.

These methods are for convenience only—you're free to use the standard C routines instead. However, any memory you allocate with one of ATtACK's allocation methods must be freed with `atk_free()`, and any memory you allocate with the C routines must be freed with the standard-C `free()`.

Note that since these methods require a session object, they are not available to analysis code, only to the instrumentation tool.

---

## A Simple Profiler

Our sample program for this lesson is an improved version of the previous lesson's program. Unlike ProcCount, this new program, SimpProf, *will* get used—it adds just enough features to become a genuinely useful profiler.

The simple design is still very close to that of ProcCount. Every time a procedure executes, a counter for it is incremented. However, now we add a new metric: Within each procedure, a counter at the start of every basic block increments an instruction counter.

Thus, we not only gather the number of times each procedure is called, but also the number of instructions that are executed within that procedure. This solves ProcCount's chief failing: Procedures that are called infrequently but have large loops, such as the typical game-loop function, no longer “fall through the cracks.”

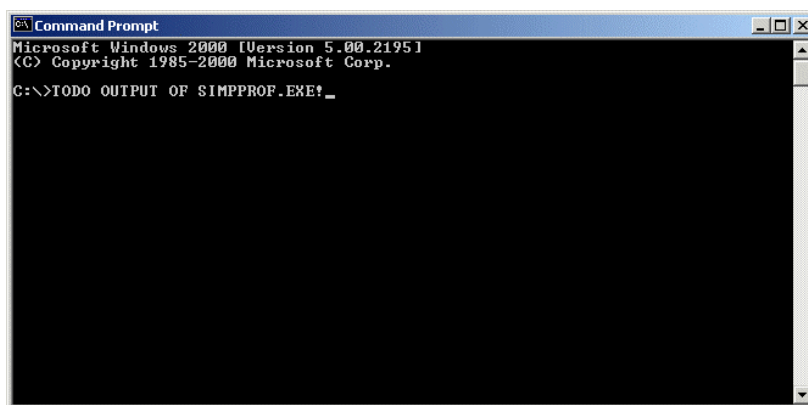
## Building and Running the Program

SimpProf is one of the sample programs that came with ATtaCK. From the main folder where ATtaCK was installed, go to Examples\SimpProf. The instrumentation tool's project, `simpprof_inst_ps2.mcp`, is located in the Inst folder, while the analysis code is in the `simpprof_anal_ps2.mcp` project within the Anal folder.

Open both those projects in CodeWarrior, and make each of them. (It doesn't matter what order you make them in.) Most likely they're already up to date, so the make shouldn't take any time.

To run the tool outside the debugger, which is easiest, just open a command prompt. The syntax is just `simpprof appname`. The results are lengthy and get dumped to `stdout`, so you probably want to redirect that with `simpprof appname > outfile`.

Assuming everything works, the application will launch and run on the target system immediately. Hit ENTER on the host to halt the target and read the data.



*Figure 04-01: SimpProf Output*

Now that we know the program works, let's look at *how* it works.



## ***SimpProf: Navigation***

### **simpprof\_inst.c**

Open up `simpprof_inst.c`, which contains all the instrumentation tool's code. You can see immediately that this is a much more sophisticated program than what we've been dealing with to date. A header file! Structures! Forward declarations! Oh my!

Ignore the `simpprof.h` header file for the moment, and let's proceed. The first thing the code does is declare its state variables, putting them all in a structure called `rundata_t`. This is used by the event handler, which we'll come to by and by.

Next we see forward declarations for the functions that do all the work. You can guess what `Instrument()`, `RunIt()`, `HandleEvt()` and `PrintData()` do. `cmpSort()` is a comparison routine required by `qsort()`, used to make the output more user-friendly, and `ClearData()` is used to reset the target's profile data. We'll see all those functions later.

### **main()**

Moving on into `main()`, we find the same boilerplate code that we know and love from Lessons 02 and 03. We're going to see this code nine more times before the course is through.

This program reads the target application name off the command line, as usual. There's an extra argument this time, the name of a procedure. As we'll see later, the analysis code will halt the application for the profile data to be read whenever the named procedure is called.

Next, we allocate a buffer for the `pdatas` array. This array contains one element for every instrumentable procedure in the target application; by dynamically allocating it, we know that the buffer is large enough without being wasteful.

Each element of this array is a `procdat_t` structure. This structure is defined in `simpprof.h`, so jump to that for a moment. There we see that `procdat_t` stores the profiling data for a single procedure, in two members: a 64-bit counter for the number of calls to the procedure, and a 64-bit counter for the total number of instructions executed within that procedure.

There's also a string constant for this structure's ATtaCK signature. That's something we haven't seen before. ATtaCK allows you to declare structures for use in instrumentation calls. Structures like this are a much better way to store profiling data than the alternative, parallel arrays. To use a structure, though, you have to declare it to ATtaCK as well as the compiler—just as to use an analysis routine you had to declare it to ATtaCK and the compiler. You can probably guess how that works, but we'll cover it in detail next lesson.

For now, take the `procdat_t` structure as a given and let's move on. Armed with our profile-data buffer, we call `Instrument()`.

## Instrument()

Apart from being a separate function rather than a section of `main()`, this is basically the same navigation and instrumentation process we've already seen, with a couple of new features.

First, we declare the `procdat_t` structure for use by ATtaCK, naming it "ProcDat." Next we declare a prototype for the `Initialize()` analysis routine. This routine takes an array of `ProcDat` structures. Arrays of structures work exactly like arrays of basic types. We'll cover structures in depth next lesson, but you're probably already getting an intuitive feel for how they work.

We then use our prototype to insert a call to `Initialize()` at the start of the program. Since it takes an array of `ProcDat` structures, we pass the instrumentation call the size of the array and the address of its first element.

Next, if the user specified a function name on the command line, we look up that name in our program. Note that in the ATtaCK model, C function names are more akin to entry points than procedures, so that's what we look up and instrument here. If the entry point exists, it receives an instrumentation call to the `Report()` analysis routine.

Finally, we enter a traditional iterate-and-instrument loop, running through all the images, then all the procedures within an image, then all the basic blocks within a procedure. Each procedure receives an instrumentation call to `CountProc()`, passing it the procedure's numeric ID. Each basic block receives a call to `CountBlock()`, passing it the procedure's ID and the number of instructions in that block.

After each procedure finishes, we release its basic-block iterator. After each image finishes, we release its procedure iterator and call `atck_img_write()`. When the image loop is done, we release the image iterator and call `atck_finish_write()`. That method gives us back a handle to our newly instrumented application, which we return to the caller.

The caller's next stop will be to pass that handle to `RunIt()`, which will download and run the instrumented application. Before we go there, though, let's look at what the analysis code does.

## ***SimpProf: Instrumentation and Analysis***

### **simpprof\_anal.c**

Open `simpprof_anal.c`, part of the `simpprof_anal_ps2` project. Even though this is a much better profiling tool than last lesson's `ProcCount`, the analysis code is really not much more involved.

First, the file includes the `simpprof.h` header, so that both the host and target are using the same definition of the `procdat_t` structure.

## Initialize()

The Initialize() function receives a pointer to an array of these structures and stores that pointer in a global variable. Remember, even though the tool passed both the size and address of the array to the instrumentation tool, we only receive the address.

## CountProc()

The CountProc() routine receives a numeric procedure ID and uses that to index the pProcDats[] array, it then increments the ncalls member of the indexed element. This is the exact same process used in ProcCount; the only differences are that the array is now dynamic and the counters are members of a structure rather than stand-alone integers.

## CountBlock()

CountBlock() receives the same numeric procedure ID as CountProc(), plus the number of instructions in the block—it increments the icount member of the indexed structure by the number of instructions. The profile data will thus show us both the number of times each procedure was called and the total number of instructions executed within that procedure.

Notice that we've instrumented basic blocks, not individual instructions. By the definition of a basic block, we know that if the block is entered, every instruction in that block *will* get executed. Thus, we can simply increment the instruction counter at the top of the basic block.

Without the concept of basic blocks, we'd have to instrument every individual instruction. You can imagine the effect that'd have on performance.

Now, it'd be nice to know *which* basic blocks get executed, rather than just the total instruction count. Knowing which procedure consumes the most time definitely helps, but we'd also like to know *why* that procedure executes so many instructions—for instance, is there a loop that runs more often than we expected? So there's definitely room for some simple improvements here, which we'll look at in the assignment.

## Report()

At first glance, this routine doesn't seem to live up to its name: It just stops execution. Remember, though, that the analysis code cannot talk to the host; the host must talk to (i.e., read from) it. The only way analysis code can get the host's attention is by stopping. This raises an event on the host, who (presumably) responds to that event by reading the profile data.

And with that, let's go back to the instrumentation code and see how it executes and analyzes the application.

## ***SimpProf: Execution and Output***

### **simpprof\_inst.c Again**

#### **RunIt()**

First, this function looks up the address of the analysis code's pProcDats variable. That address, along with the other state information that will be required to read the profile data from the target, is stored in the rdata structure.

Next, RunIt() goes through the standard boilerplate of connecting to the target. It specifies HandleEvt() to be the callback event handler, and passes in a pointer to the rdata structure. ATtaCK will pass this pointer to HandleEvt() along with any events that get generated.

The function then downloads the application and immediately starts it running. The fgets() library function waits for user input, which is a good way to set up an idle loop. If the user hits ENTER, the target is stopped, which results in the event handler being called; if the user hits Z before ENTER, the event handler will clear the profile data buffer, otherwise it will print the buffer's contents.

If the user hits X before ENTER, the loop immediately ends and the target is disconnected. This automatically kills the running program, which generates one last call to the event handler.

Finally, RunIt() returns to main(), which goes through the standard end-of-session cleanup boilerplate.

#### **HandleEvt()**

The event handler receives a handle to the event, a pointer to the rdata state structure and a dummy pointer, which gets discarded.

If the event is of type ATCK\_TEVT\_STOPPED, the handler gets the running-program handle from the event. Based on a flag set by the user-input loop in RunIt(), the handler either calls ClearData() to wipe the target's profile buffer or PrintData() to read and display the buffer.

The other events are just diagnostic or progress messages, which can be displayed or not as you see fit. In this particular tool, we chose to display the progress messages but not the diagnostic ones.

#### **PrintData()**

The first thing PrintData() does is find out how many instrumentable procedures the original application has—which is, of course, the number of elements in the profile data array.

Then, by reading a “buffer” of a single address, it gets the contents of the target's pProcDats pointer. It uses that address in turn to read the contents of the target's profile data array.

Next, `PrintData()` allocates a temporary array of `sort_t` structures. This structure holds a handle to a procedure and a pointer to that procedure's profile data. A standard iteration loop fills in the structures, and then the array is sorted in descending order of instruction counts using the standard C library routine `qsort()` and the user-defined comparison routine `cmpSort()`.

`PrintData()` then runs through this sorted array, printing out the name, number of calls, total instruction count and percentage of the program's entire instruction count for each procedure.

Finally, `PrintData()` releases the temporary array and restarts the target.

### **ClearData()**

Using the same process as `PrintData()`, `ClearData()` gets the address of the target's profile data buffer and writes an array full of zeroes to it, then restarts the target.

### **Almost There!**

With that out of the way, we've covered 90% of the documentation in 60% of the course. We still need to go over the details of dynamic arguments, and there are some register-usage attributes of the instrumentation object to discuss.

After that, though, the rest of the course will just focus on nine more sample programs. Each of these programs solves real-world problems, and they're all as useful as you'll find `SimpProf` to be.

So without further ado, let's continue to Lesson 05: Designing Analysis Tools.

## Lesson 04 Assignment

SimpProf counts the number of instructions executed within each procedure, by incrementing the procedure's counter by the number of instructions contained within each basic block that gets executed. In other words, we know *how many* instructions get executed in a procedure, but not *which* instructions.

That's usually good enough, but there are times when it's inadequate. This is one of those times: Your AI pathfinding routine, `WalkMap()`, is executing more than twice as many instructions as it should. Hack SimpProf, as quickly and easily as possible, to give you the information you need.

### Hint:

You'll find the methods `atck_ent_byname()` and `atck_ent_proc()` very useful.

### Answer:

As with any coding exercise, there's more than one right answer, but your solution should look pretty similar to mine. Basically, you need to instrument each basic block within `WalkMap()` with a special version of `CountBlock()` that increments a counter for the block in addition to the counter for the procedure.

Remember, what we're aiming for here is a hack: You're not trying to improve SimpProf, you're trying to improve `WalkMap()`! So take as many shortcuts as you can. In particular, you know roughly how many basic blocks `WalkMap()` has, so create a static array in the analysis code to hold the basic-block counters rather than dynamically allocating one. Reading and displaying the contents of this array is trivial, so we'll skip that.

Here's the code I added to `Instrument()`, right before the comment that begins "This is the main part of the instrumentation."

```
pent = atck_ent_byname(pprog, "WalkMap");
if (pent)
{
    int ibb = 0;

    pproc = atck_ent_proc(pent);
    pproto = atck_analproto(pprog, "void
                                HackCountBlock(uint32,uint32)");
    pbbit = atck_proc_bbit_new(pproc);
    while (pbb = atck_bbit_next(pbbit))
    {
        atck_bb_callbefore(pbb, pproto, ibb,
                           (atck_uint32_t)atck_bb_ninst(pbb));
    }
}
```

```

        ++ibb;
    }
    atck_bbit_free(pbbbit);
}

```

And here's the new analysis code:

```

atck_uint32_t  BlockCounters[25];

void HackCountBlock(atck_uint32_t ibb, atck_uint32_t
numInst)
{
    BlockCounters[ibb] += numInst;
}

```

I also added a tiny loop to `Initialize()`, in order to clear the array to zero at the start of the program:

```

int i;
for (i = 0; i < 25; i++)
    BlockCounters[i] = 0;

```

## Lesson 04 Quiz

1. Where can you find the file referred to by `<syscfg>`?
  - A. The directory containing your ATtaCK license
  - B. The Windows system directory
  - C. The directory containing `atck.lib`
  - D. The CodeWarrior compiler directory
2. True or false: An ATtaCK tool can monitor the status of the target system without using an event handler.
  - A. True
  - B. False
3. True or false: The event-handler arguments `devID` and `progID` must be pointers to structures.
  - A. True
  - B. False
4. True or false: The ATtaCK config-file system only lets you read, not write, files.
  - A. True
  - B. False
5. True or false: The only way to create a new `iprogram` handle is by calling `atck_iopen()`.
  - A. True
  - B. False
6. Which of the following is *not* an option expected by `atck_connect()`?
  - A. Ip
  - B. Port
  - C. HostioDrive
  - D. devID
7. Which of the following functions may *not* be called from within an event handler?
  - A. `atck_stop()`
  - B. `atck_kill()`
  - C. `atck_continue()`
  - D. None of the above



8. When an application has locked a mutex using `atcktarg_lock()`, how can the application be halted or interrupted?

- A. By system interrupts
- B. By the host stopping the target with `atck_stop()`
- C. By the host stopping the target with `atck_kill()`
- D. By another thread locking that same mutex

9. True or false: Every member in an ATtaCK structure declaration needs to be named.

- A. True
- B. False

10. True or false: You are not required to use the special ATtaCK memory-allocation methods such as `atck_malloc()`.

- A. True
- B. False

## Lesson 05: Designing Analysis Tools

---

By this point, you're proficient with using and modifying the existing ATtaCK sample tools. In our fifth lesson, we'll dig deeper into designing your own analysis code. We'll discuss some basic principles of code analysis and look at all the information your analysis code can gather. Finally, we'll examine PS2Counter, an extremely useful tool that uses the EE's hardware performance counters to quickly collect accurate performance information.

### *The Road Ahead*

Between what you've learned so far, and keeping the documentation handy, you should be able to look at any ATtaCK tool and figure out what it's doing. Really, though, the point of ATtaCK is to allow you to create custom tools that answer specific questions you have about your code. To get the most out of ATtaCK, you need to go beyond learning how ATtaCK tools work and master how to design new tools of your own.

Now, in a live course, or even a one-on-one interactive version of this online course, that's exactly what we'd spend the rest of our lessons working on. We'd stop here and go straight to your own code—you'd tell me what problems you're trying to solve, and I'd show you how to use ATtaCK to, well, attack them.

Unfortunately, from where I'm currently writing—April 2001, on a rooftop in Israel—I can't know what your particular code problems are. So rather than just stop the course short, I'll keep going as I have been, covering general-purpose tools that answer questions everybody tends to have:

- “How often does my program read from the cache rather than memory?”
- “Which functions should be inlined?”
- “Which branches should be reversed?”
- “Where's that \*^!% @# memory leak coming from?”

There are seven of these general-purpose tools to come, one in this lesson and three each in the next two lessons. Each of them is a full-fledged, standalone tool, and you might find the whole suite of them worth ATtaCK's price tag.

Before we get stuck forever in the land of general-purpose tools, though, we'll cover the remaining handful of ATtaCK features you haven't yet seen: dynamic arguments, user-defined data structures and instruction register-usage attributes.

Before *that*, though, we'll talk about some principles of code analysis and how they can help you design your own ATtaCK tools.

### *Principles of Code Analysis*

What follows are some guiding principles of code analysis that I've found useful over the years. I don't pretend these to be jewels of crystallized wisdom, just

handy rules of thumb. I'm sure many of you taking this course are senior programmers to whom these will be old hat. You have to read this page anyway, because it'll be covered in the quiz.

On the other hand, a lot of you are probably junior programmers. "The lead programmer is way behind schedule; let's have the new guy right out of CS school run the profiler"—does that sound familiar? If you're in that situation, I can't drag you out of the deep end, but I can at least throw you a rope.

## The Role of Analysis

People think of analysis as relating to optimization, which is partially true. However, what people think of as "optimization" is more properly called "tweaking" or "fine-tuning," which is really just a form of debugging. Code analysis provides information for all forms of debugging, not just performance tuning. Your IDE debugger, for instance, is a code-analysis tool.

No matter what type of debugging task you're using it for, you have to keep in mind that analysis gives you a very low-level view into your code. It can tell you what's going wrong, but it shouldn't help you fix it. Most problems, whether they're code defects, speed problems or whatever, need to be fixed at the highest level possible.

The absolute best place to fix software defects is at the game-design level. If the game design calls for twelve robot guards per level, but your AI routines can only handle ten before becoming too slow, you have a right to ask the designer whether he can live with just ten robots per level. Programming is an extremely expensive and risky endeavor, and the less of it the game requires, the better.

Of course, every time I've made that point, at all the companies I've worked for or consulted to, it's fallen on deaf ears. I don't have any illusions that this time will be any different. So let's assume the game design is graven in stone...

The next best place to fix defects is at the code-design level. This is the land of algorithm changes, render-pipeline shortcuts and other tricks that let you cut out whole chunks of code from your program. Programmers often think of design changes only as an optimization technique, but they can address code defects as well. For instance, you might be able to fix a memory leak simply by changing how or where you allocate memory.

The function level, though, is where the typical programmer feels most comfortable. This is the land of inline functions, unrolled loops and the like. You might well lack the authority to make changes above this level. For that matter, many problems can *only* be fixed at this level, since this is where they were introduced—having a brilliantly streamlined render pipeline doesn't mean you can afford to implement it with buggy code. Nevertheless, any time you're debugging at this level, a voice in the back of your mind should be questioning whether you can't accomplish the same task better through a design change.

Finally, the instruction level is where the hard-core assembly programmers get to have fun. This is the land of branch-prediction optimization, five-cycle render loops and other clever tweaks that earn you the admiration of your hacker peers.

The trouble is, while these tricks can make a fast program faster, they won't make a slow program less slow. It's the same reason why they put afterburners on F-16s rather than B-52s.

Notice that each level down brings you closer and closer to the information ATtaCK generates, because each level is closer to the actual machine code that ATtaCK measures. This is the single biggest danger of using ATtaCK: The wealth of low-level data you gather encourages you to focus your efforts on the low-level code, rather than looking at the big picture. The higher levels, though, are where you get the most return on your programming dollar. Always, *always* keep that in mind when you work with ATtaCK.

## Analysis Principles

Now that I've gotten the "with great power comes great responsibility" speech out of the way, here are the rules-of-thumb I've found most useful when analyzing code.

### The 80/20 Rule

The most interesting 80% of anything is usually contained in 20% by volume. For instance, 80% of your bugs are usually found in 20% of your code. Whenever analysis reveals a bug, you should check the portion of your program most similar to it—same author, written at the same time, performing the same task, whatever—for other bugs.

Likewise, 80% of your performance problems usually come from 20% of your code. One of the very first things you should do when performance-tuning an application is run SimpProf and make note of the top 20% of the procedures by instruction count. Those routines should receive the bulk of your attention.

You have to actually measure the entire program, however. While 20% of your code needs 80% of the work, don't assume that you know *which* 20% that is.

### Measure, Don't Guess

It's amazing how often this slips by people (including me, I have to admit): If your program has a problem, then *obviously* your assumptions were wrong somehow. (I guess you could be incompetent or malevolent—I've encountered both!—but that's a story for a different time...) So when you're performing analysis—that is, when you're trying to figure out what mistake you made—you should be especially quick to challenge your assumptions and test your preconceptions. The possibilities that you dismiss out of hand are the very first ones you should use analysis to test.

I can't count the number of times I've been helping one of my programmers debug his code and heard him say "That code can't *possibly* be the problem." That's always my cue to say "Well, let's check it anyway, just to be sure"—and more often than not, that's exactly where the bug is. This is one of the reasons why simply having an extra person helping you debug or optimize is so useful—each of you can check the other's assumptions.

## The Heisenberg Uncertainty Principle

The Heisenberg Uncertainty Principle is a rule of quantum physics that, broadly stated, means that it is impossible to measure something without interacting with it. It's true of analysis, too: The mere act of measuring performance will invariably reduce it. Likewise, there are some bugs that go away when you run the app under the debugger, or that go away when you simply recompile. ATtaCK can help you track those down, since it uses a different mechanism than the debugger, and doesn't require you to recompile your program in order to insert diagnostic code.

The Heisenberg Uncertainty Principle tells physicists that there's an ultimate limit to how accurate any measurement can be. That goes for ATtaCK too—at a certain point, the performance information you get from the tool is less than the performance penalty of the instrumentation code. That's the point to stop tweaking—perhaps you could make your program faster, but there is such a thing as “good enough.”

## Designing Analysis Tools

Notwithstanding the fact that I just spent four entire lessons talking about nothing else, actually programming ATtaCK tools is very simple. The challenge is in *designing* analysis tools. The perfect ATtaCK tool has three features:

- It is *focused*, collecting exactly the data you need, so that you get the answer to your question without being overwhelmed by information.
- It is *lightweight*, so that the analysis process doesn't distort your results or interfere with playtesting.
- It is *simple*, so that you can focus your efforts on actually solving problems rather than writing and maintaining analysis tools.

### Focused

When you start designing (or using) an analysis tool, stop and ask yourself what you'll do with the information it collects. In the case of a profiler, the answer is easy: You'll use the profile to show you which areas of the program require the most optimization effort. As your tools get more specific and detailed, however, you should become more skeptical about their value.

For example, one of the tools we'll see later estimates how often your code fetches data out of the cache rather than from memory. Let's say you run that tool and find out that your program misses the cache 74.4% of the time. Now what? You've asked a valid question, and you've gotten an informed and accurate answer, but if you don't know what to do with that answer you've just wasted your time.

Instead of asking “How often does my program miss the cache?” a better approach would be to ask (for instance) “How often does my *AI pathfinding*

*routine* miss the cache?” By limiting the scope of the question to a single function, you make the answer more comprehensible and thus more useful.

Better still would be to ask first “What data does my AI pathfinding routine fetch most often?” if you don’t already know that answer. This enables you to be more specific still: “How often does my AI pathfinding routine miss the cache *when fetching map data*?” Now when you get the answer “74.4%,” you know that you should probably modify your search algorithm to walk the map by rows rather than columns.

It’s seductive to simply collect as much information as possible. That requires less planning in the tool, and seems to make the tool more “reusable,” thus reducing your work down the road. However, writing the tool is the easy part. The hard part—and the part you get paid for, by an amazing coincidence—is solving real problems in your application. That cause is best served by special-purpose tools that deliver narrow, focused information.

## Lightweight

Here’s a scenario I’ve seen play out several times: Lead programmer discovers the value of a profiler, memory-leak checker, code validator or some other analysis tool. Lead programmer convinces management to buy copies of the tool for every programmer, so that they can all run it all the time. Within a week, the tool is collecting dust on the shelves, as the programmers stop using it so that they can “get some actual work done.”

Does that sound at all familiar? The problem here is that Heisenberg Uncertainty Principle again: Measurement invariably interferes with the thing being measured. In the case of analysis tools, if the cost they impose in programmer or playtester time is too great, then the programmers and playtesters will just revolt and not use them.

When designing analysis routines, you should move as much of the processing out to the host as you can, so that the game remains playable. We’ve already seen how a tool can keep all the context info on the host, so that the analysis code only deals with ID numbers for procedures. More generally, you should pass as little information as possible to analysis routines. This ties in to the previous point about collecting narrow information: The more focused your tool, the less impact it will have on the target application.

Target clock cycles aren’t the only resource your tool might waste, however. An analysis tool that must instrument the program each time it runs, and that takes ten minutes to do the instrumentation, wastes tester time—better to break the instrumentation code out from the download and execution code, so that the tester only has to sit through the time-consuming instrumentation process once. Relatively small and quick changes like that can have big payoffs in usability. Talk to the people who are actually using your analysis tools and ask them for feedback.

## Simple

ATtACK does a lot of work for you. Take advantage of that—write your tools in the simplest, most direct way possible, using the library code as much as you can. Especially when working with ATtACK’s iterators, the most intuitive approach to a task may well require the most work, while a less “natural” or “elegant” solution will let the API handle everything for you.

For example, given an instruction handle, you can get the handle of the containing basic block, but there’s no convenient way to get the handle of the next instruction in that block. If you find yourself needing to write code to do that, then you’ve probably made a mistake—you’ve written a function that works with instruction handles, when really you should write one that works with a basic block or an instruction iterator. (C++ programmers should recognize this as one of the rules of OOP design: If you find yourself needing `friend` access, you’ve probably designed your classes poorly.)

Aside from forcing yourself to use the ATtACK API as much as possible, write in the style that’s most natural for you. If you’re most comfortable writing uncommented code with one-letter variable names—well, it’s not a habit I want to encourage, but go ahead and indulge yourself if that’s what it takes to get the tool written faster. If maintenance turns out to be an issue, you can clean things up later, when you don’t have a big problem looming over you.

It’s often more efficient to write a sloppy but customized tool focused precisely on the task at hand than to sift through the output of a more general-purpose, well-written one. When you’re jotting down notes during a debugging session, neat handwriting and proper grammar are less important than getting the information you need; ATtACK tools are just the software equivalent of that scratch paper.

## Dynamic Arguments

Okay, so much for the guru-on-the-mountaintop bit. Now we’ll introduce dynamic arguments and look at the last remaining API functions. We’ll finish the lesson with a look at an analysis tool that takes advantage of a super-elite secret hardware feature of the EE.

Dynamic arguments are values known only at the application’s runtime, not the tool’s runtime. These are used to communicate information from the running application to the analysis routines, and are in fact the *only* way to do so—analysis code does not have access to the running program, and can only act on information passed to it via instrumentation calls.

The most common dynamic arguments are *registers*. Technically, an analysis routine could simply read registers itself rather than being passed them. However, that would require writing the routine in assembly, which many programmers don’t know. More importantly, registers might get modified in between the instrumentation call and the analysis routine. (This is nearly guaranteed to happen in the case of the link register GPR31!)

Dynamic arguments can also pass *addresses*. An address argument can be used to identify the memory location accessed by a load or store operation. It can also be used to identify the instruction address targeted by a branch, call or return operation.

“But wait!” you say. “I already know the targets of branches and calls, thanks to methods like `atck_call_targaddr()`. Why do I need them at runtime?” Well, for two reasons. First, you might *not* know them in advance—you don’t know the targets of C++ virtual method calls until runtime, for instance.

Second, the act of instrumenting the program might change all of those addresses. ATtaCK inserts code into the instrumented program, and that code has to go somewhere. Actually, though, on MIPS platforms like the PlayStation 2, all instructions are the same size, so ATtaCK *doesn’t* have to change any addresses. Still, on other platforms it might, so don’t rely on this beyond the PlayStation 2.

Similarly, the tool can also simply pass in a static address from the original application, such as the address of a function, which gets translated into the corresponding address from the instrumented application. Technically this isn’t really a dynamic argument, since it’s known before application runtime, but it uses the same syntax as dynamic arguments.

For applications with multiple images, a simple address is not sufficient, since each image may have its own address space. ATtaCK can thus pass *image ID numbers* to complement addresses. This won’t happen automatically—you have to explicitly ask for both the address and the image ID, as we’ll see in a moment.

Finally, there’s a special dynamic argument available called the *condition-taken flag*. This is used with instrumentation added after a conditional store or before a conditional branch to tell the analysis routine whether the operation was taken. Of course, the PlayStation 2 doesn’t *have* conditional stores, so this is really just for identifying whether a conditional branch was taken or skipped.

## C Declarations

As with static and array arguments, your analysis routines see dynamic arguments as they would any other arguments: as basic types. Specifically, image IDs and condition-taken flags come through as `atck_uint32_t`, integer registers as `atck_uint64_t`, floating-point registers as `atck_float32_t` and addresses as `atck_addr_t`.

As an example, let’s create an analysis routine for logging conditional branches. The routine takes a static integer (used by the instrumentation tool to identify this particular branch), the target address and image of the branch, a condition-taken flag, and the contents of the register tested for the condition. The C declaration of this routine would look like this:

```
void LogBranch(atck_uint32_t id, atck_addr_t destaddr,
               atck_uint32_t destimgid, atck_uint32_t
               taken,
               atck_uint64_t value);
```



## ATtaCK Declarations

Likewise, the ATtaCK declaration for a dynamic argument specifies the argument's type, modified to indicate that the argument is dynamic rather than static. The ATtaCK declaration does *not* specify what value the argument will be used to pass, however—that's done by the instrumentation call.

Registers are declared as `regv64` and `fregv32` for integer and floating-point registers respectively. Runtime addresses are declared as `valaddr`; instrumented-application addresses are declared as `instaddr`. Image IDs and condition-taken flags are declared as `valuint32`.

The ATtaCK declaration of our example `LogBranch()` routine would look like this:

```
"LogBranch(uint32, valaddr, valuint32, valuint32, regv64)"
```

The real work of dynamic arguments is done at the instrumentation call.

## Instrumentation Calls

The declaration tells ATtaCK to expect a dynamic-argument flag instead of an actual value in the instrumentation call; the value you then pass to the instrumentation call specifies which dynamic argument to use. For instance, the declaration specifies that you'll be passing a register; the instrumentation call itself identifies the particular register to pass.

## Registers

Remember the `atck_reg_t` enumerated type from way back in Lesson 02? Now you get to see what it's for. In your instrumentation call, you pass a value of this type to indicate which register you want to pass to the analysis routine. At runtime, ATtaCK replaces this value with the contents of the specified register. Remember that you need to use `regv64` for the integer registers, `fregv32` for the floating-point ones.

You can only pass registers via instrumentation calls added to instructions. The contents of a register are undefined anywhere else in the program. If you really need to analyze the contents of registers at the start of a procedure (for instance, if you want to log the arguments passed to the function), use `atck_inst_callbefore()` on the very first instruction of the procedure.

If you pass a register via `atck_inst_callbefore()`, your analysis routine receives the register's value before any changes made by the instruction. If you pass it via `atck_inst_callafter()`, your routine gets the value after all the instruction's side effects. If the value wouldn't normally be immediately readable—for instance, in the case of a load from memory—ATtaCK automatically inserts enough delay cycles to ensure the value is available. (Actually, on the PlayStation 2, the processor will itself insert the delay cycles, but the effect is the same.) Be aware of the performance hit this will cause!

We'll look at registers again next page.

## Runtime Addresses

To pass the address of data loaded or stored by an instruction to an analysis routine, you pass the flag value `ATCK_EFFADDR` to the instrumentation call. This value is replaced at runtime with the memory address affected by the instruction. This flag only has meaning when passed to `atck_inst_callbefore()`; it won't work with `atck_inst_callafter()`, and obviously it only applies to instruction instrumentation.

Similarly, you can pass the target address of a branch instruction using the `ATCK_TARGADDR` flag with `atck_inst_callbefore()`. You can also pass the target of a call using the `ATCK_TARGADDR` flag with `atck_call_callbefore()`.

Finally, you can pass the address to which a procedure is returning using the `ATCK_TARGADDR` flag with `atck_proc_callafter()`.

## Runtime Integers

In any situation where you could use `ATCK_EFFADDR` or `ATCK_TARGADDR`, you can also use `ATCK_EFFIMG` and `ATCK_TARGIMG` to identify the specific image containing the passed address. On the PlayStation 2, where you can only instrument single-image applications anyway, this won't come into play.

You can pass the `ATCK_TAKEN` flag into `atck_inst_callafter()` on a store operation to find out whether the store was actually executed. It is replaced with the value `ATCK_TRUE` if the store was executed, or `ATCK_FALSE` otherwise. This is only available with instructions, and must be called after the instruction. Note that the EE doesn't have conditional stores, so while you can still use `ATCK_TAKEN`, it will always be `ATCK_TRUE`.

Between image IDs and conditional-store flags, runtime integers may seem useless. Well, there's one more situation where you can use `ATCK_TAKEN`: before a branch, using `atck_inst_callbefore()`. If the branch is conditional and not taken, this value will be passed as `ATCK_FALSE`; unconditional or taken conditional branches pass `ATCK_TRUE`.

## Application Addresses

Application addresses may be declared and used with any instrumentation call. Simply pass an address from the original application into the call. As long as the address is contained within the uninstrumented application, it is replaced with the corresponding address from the instrumented application. If ATtaCK can't recognize the address as part of the original application, then the address is passed unchanged to the analysis routine.

To conclude our example, here's what one instrumentation call to our `LogBranch()` analysis routine might look like:

```
atck_inst_callbefore(pinst, plogbranchproto, instid,  
                    ATCK_TARGADDR, ATCK_TARGIMG,  
                    ATCK_TAKEN, ATCKMIPS_REG_GPR6);
```

The complete set of dynamic arguments is summarized in Table 05-01.

| <i>Argument Type</i>        | <i>C Type</i>                      | <i>ATtaCK Type</i>   | <i>Expected Value</i>                      |
|-----------------------------|------------------------------------|----------------------|--|
| <b>Register</b>             | atck_uint64_t or<br>atck_float32_t | regv64 or<br>fregv32 | Register flat (e.g.,<br>ATCKMIPS_REG_GPRO) |
| <b>Affected Address</b>     | atck_addr_t                        | valaddr              | ATCK_EFFADDR                               |
| <b>Affected Image</b>       | atck_unit32_t                      | valuint32            | ATCK_EFFIMG                                |
| <b>Target Address</b>       | atck_addr_t                        | valaddr              | ATCK_TARGADDR                              |
| <b>Target Image</b>         | atck_uint32_t                      | valuint32            | ATCK_TARGIMG                               |
| <b>Condition-Taken Flag</b> | Atck_uint32_t                      | Valuint32            | ATCK_TAKEN                                 |
| <b>Application Address</b>  | Atck_addr_t                        | Instaddr             | Address from uninstrumented<br>application |

*Table 05-01: Dynamic Arguments*

## Restrictions

As you can see in the text above, not all calls can take all dynamic arguments—they’re mostly restricted to calls added to instructions. In fact, they’re not even valid for all instructions.

Most of the restrictions are common sense—you can’t use `ATCK_EFFADDR` with an instruction that only affects registers. Some are more obscure. For instance, you can’t use `ATCK_TAKEN` with the EE’s coprocessor branch instructions. There may even be context-sensitive reasons why a particular instruction may not permit an instrumentation call that would be valid for an identical instruction somewhere else.

To find out whether an instruction can take a given dynamic argument, you can check the following attribute:

```
atck_bool_t atck_inst_isallowed(atck_inst_t* self, int code)
```

This is a method of the instruction object, so of course a handle to the object is passed as the first argument. The second argument is the dynamic-argument flag you want to test: `ATCK_EFFADDR`, `ATCK_TARGADDR`, `ATCK_EFFIMG`, `ATCK_TARGIMG` or `ATCK_TAKEN`. The return value is either `ATCK_TRUE` if the argument is allowed in this context, or `ATCK_FALSE` otherwise.

You don’t *have* to check to see whether an instrumentation call is permitted. You can simply insert the call. If the call or its arguments are forbidden, ATtaCK will halt and issue a diagnostic message. That wouldn’t be acceptable behavior in an actual application, or in a robust, reusable analysis tool, but might be just fine for a throwaway program.

## Working with Registers

Way, way back in Lesson 02, I mentioned two additional attributes of the instruction object: which registers the instruction reads and writes. Since an instruction will often touch multiple registers, these attributes have to return multiple values. And since there are more than 64 registers on the EE, returning

bit flags in an unsigned integer isn't an option. Instead, ATtACK uses a special object called a *register set*.

## Register Set Object

The register-set object, `atck_regs_t`, performs the same function that a set of bit flags would. In fact, on a platform with very few registers (like, say, the x86), the register-set object would probably just be a wrapper for an integer containing bit flags—but that sort of implementation detail is hidden from you.

But since it's essentially simulating a large collection of bit flags, the register-set object provides methods to give you the same abilities you'd have with bit flags: toggling individual bits on and off; turning all the bits on or off; testing an individual bit; counting the number of set bits; and combining sets with AND and OR.

## Managing Register Sets

You're responsible for allocating and releasing all the register sets you use. As with most objects in ATtACK, this is done using `new` and `free` methods:

```
atck_regs_t* atck_regs_new(atck_prog_t* self)
void atck_regs_free(atck_regs_t* self)
```

`atck_regs_new()` is a method of the program object—that's the object that knows about the target processor, and thus knows what the range of possible registers is. It returns a handle to an empty register set, which will eventually be freed using its `atck_regs_free()` method. If you don't free up a register-set object, it will get freed for you automatically when you close the session using `atck_endsession()`.

## Adding and Removing Registers

```
void atck_regs_add(atck_regs_t* self, atck_reg_t reg)
void atck_regs_rem(atck_regs_t* self, atck_reg_t reg)
```

These methods add or remove the specified register from the register-set object. The *reg* argument is one of ATtACK's register ID constants. If you were using bit flags, these methods would be exactly equivalent to `self |= reg` and `self &= ~reg`.

```
void atck_regs_addall(atck_regs_t* self)
void atck_regs_remall(atck_regs_t* self)
```

These methods add or remove *all* registers from the set. If you were using bit flags, these methods would be exactly equivalent to `self = 0` and `self = -1`.

## Testing Registers

```
atck_bool_t atck_regs_ismem(atck_regs_t* self, atck_reg_t reg)
```

This method tests whether the specified register is part of the set, returning `ATCK_TRUE` if it is, `ATCK_FALSE` if it isn't. If you were using bit flags, this method would be exactly equivalent to `(self & reg)`.

```
unsigned atck_regs_num(atck_regs_t* self)
```

This method returns the total number of registers contained in the set. This method doesn't have a convenient equivalent with bit flags; I always wound up writing one, and I guess the ATtACK designers did too!

## Combining Register Sets

```
void atck_regs_addset(atck_regs_t* self,
                     atck_regs_t* src1, atck_regs_t*
                     src2)
```

This method makes the specified register set contain the union of *src1* and *src2*. Any of the three arguments may be the same register set. If you were using bit flags, this method would be exactly equivalent to the statement `self = src1 | src2`.

```
void atck_regs_intset(atck_regs_t* self,
                     atck_regs_t* src1, atck_regs_t*
                     src2)
```

This method makes the specified register set contain the intersection of *src1* and *src2*. If you were using bit flags, this method would be exactly equivalent to the statement `self = src1 & src2`.

```
void atck_regs_remset(atck_regs_t* self,
                     atck_regs_t* src1, atck_regs_t*
                     src2)
```

This method makes the specified register set contain every register from *src1* that is not in *src2*. If you were using bit flags, this method would be exactly equivalent to the statement `self = src1 & ~src2`.

## Register-Usage Attributes

```
void atck_inst_inregs(atck_inst_t* self, atck_regs_t* dest)
void atck_inst_outregs(atck_inst_t* self, atck_regs_t* dest)
```

These attributes of the instruction object identify which registers the instruction reads from and writes to, respectively. You must first allocate and pass in a register-set object to contain the result.

The existing contents of the set are *not* cleared; the affected registers are simply added to the set. If you were using bit flags, these methods would be exactly equivalent to `dest |= atck_inst_inregs(self)` and `dest |= atck_inst_outregs(self)`.

And that's it—we're done with the entire ATtACK API! For the rest of the course, we're just going to cover sample programs. So let's get started...

## The EE Register Set

ATtACK uses an enumerated type, `atck_reg_t`, to hold identifiers for all the EE registers. Unlike most of the ATtACK API, this type is platform-dependent, and requires you to include the processor-specific header in addition to the normal `atck.h` header. For the PlayStation 2, this file is `atcktargps2.h`.

If you're going to work with registers in ATtACK, you really need to get yourself up to speed on MIPS assembly language. Here's a cheat sheet to get you started, however.

- `ATCKMIPS_REG_GPR0` to `ATCKMIPS_REG_GPR31`

These are the standard 32 64-bit integer registers. By convention, some have specific meanings:

- `GPR0`, called `$0` in source code, always contains the value zero.
- `GPR4` through `GPR7` are called `a0` to `a3` in source and are used to pass arguments into functions, in order.
- `GPR2` (and `GPR3`, if necessary), called `v0` and `v1`, hold the return value.
- `GPR16` through `GPR23` are used to hold register variables; all subroutines are supposed to preserve the values of these registers.
- `GPR29` is the stack pointer, although the stack isn't used as much as on x86 chips—arguments are passed in registers, and even the return address of a function is stored in a register (`GPR31`, to be exact).
- `GPR30` holds a pointer to the function's local variables (its stack frame).
- `ATCKMIPS_REG_SA`

This register is used by several of the shift instructions to specify the number of bits to shift.

- `ATCKMIPS_REG_HI` and `ATCKMIPS_REG_LO`

These two registers store the results of integer multiplies and divides.

- `ATCKMIPS_REG_FPR0` to `ATCKMIPS_REG_FPR31`; `ATCKMIPS_REG_ACC`

`FRP0` to `FPR31` are the EE's 32 32-bit floating-point registers, which are all general-purpose. `ACC` is the floating-point accumulator.

- `ATCKMIPS_REG_FCR0` and `ATCKMIPS_REG_FCR31`

These are the floating-point control registers.

## Measuring Performance Counter Events

The EE includes a system control coprocessor, known as `COP0`. This coprocessor controls the CPU's operation—for instance, `COP0` is used to enable or disable interrupts, manage memory paging, set debugger breakpoints, handle exceptions

and so forth. COP0 also includes a pair of “performance counters,” special registers that are automatically incremented when certain events occur.

By instructing these counters to measure events that interest us, we can, in effect, take advantage of a hardware profiler for our code. This offers two major advantages. First, the performance counters are 100% accurate—in fact, as we’ll see later on, they’re *so* accurate that their results can be confusing! Second, enabling the counters does not affect the system’s overall performance. The performance counters aren’t versatile enough to handle all your profiling needs, but they’re free, easy to work with and can greatly improve the results you get with ATtACK tools.

This lesson’s example program, PS2Counter, instruments an application to enable and read these performance counters. You can choose which two events to measure, out of a list of ten possible events. The tool also uses static analysis to refine and limit its measurements, so that they’re more useful. As a stand-alone tool, this program is mildly useful... but as a source of ideas to use in creating your own profilers, it’s invaluable.

Composed of four source files and a header file, PS2Counter is far and away the biggest program we’ve looked at so far. The easiest way to understand what it’s doing is to start with the analysis code. And the easiest way to understand the analysis code is to look at the specifics of how the EE’s COP0 performance counters work.

## EE Performance Counter Registers

COP0’s two performance counters, named “0” and “1” in a fit of charming originality, each measure two different sets of events. You specify which events to log by writing to the “performance-counter specifier” register, using an assembly command we’ll look at later. The two counters are wholly independent of each other—one can be measuring instructions, for instance, while the other measures mispredicted branches.

Once a counter has been set to log a particular event, the counter increments whenever that event occurs. If set to log instructions, say, the counter is incremented with every instruction issued. Note that at full clip, the EE can issue about 600,000,000 instructions a second, which fills up a 32-bit counter pretty quickly! (In less than eight seconds, to be exact.)

Clearly these counters can’t be used to collect data over a long period of time. Instead, you should turn the counters on just over the section of your program that most interests you and gather a short span of data. The PS2Counter tool we examine here, for instance, collects data from just a single frame. Presumably, your target application performs roughly the same tasks in the same order each frame, so you can use the profile of a single frame to draw conclusions about the rest of the program.

Once you’ve gathered the data you want, you can use other special-purpose assembly commands to read and clear the performance counters.

## Performance Counter Events

**Cycle:** Either counter may be set to count cycles. No matter what, some 294,000,000 cycles pass by every second. This may not seem like the most interesting event to measure, but it can be handy for detecting problems. For instance, “cycles per frame” is a much more precise and informative measurement than “frames per second.”

**Single issue/double issue:** The EE core tries to issue two instructions at once whenever it can, but sometimes (due to register dependency or instruction incompatibilities) it fails. Double issues are thus the default, “correct” behavior; single issues represent inefficiencies you should try to correct. Single issues may only be measured in counter 0, double issues only in counter 1.

**Branch taken/branch mispredicted:** Counter 0 can count the number of branches taken. Counter 1 can log the number of branches mispredicted. This is not merely a subset of branches taken, of course—a likely branch that was not taken still counts as a misprediction.

**I\$/D\$ miss.** Like all high-end processors, the EE uses a cache of extremely fast RAM to isolate the processor from the much slower system RAM. Cache misses, where your code needs data that’s stored in system memory rather than the cache, slow down performance dramatically, and most really high-octane optimizations are tweaks designed to avoid cache misses at all costs. Unfortunately, we don’t really have the time or space to talk about such tweaks in this course. In Lesson 07, we’ll cover a cache analysis tool that will help you diagnose cache problems, and I’ll try to give some very general tips there.

The performance counters aren’t very good at *diagnosing* cache misses, because they only tell you how many times the cache was missed—the program we’ll look at in Lesson 07 tells you exactly where your cache misses are. Nevertheless, the performance counters can help you *detect* problems. Counter 0 can be set to measure instruction-cache (I\$) misses, while counter 1 can measure data-cache (D\$) misses.

**Instruction executed:** Either counter may be set to measure the number of instructions issued. This value should always wind up being the number of single issues plus twice the number of double issues. By adding instrumentation to every basic block of your program, you can count this yourself, of course. The difference is that the COP0 performance counters do so without slowing down the application.

Note, however, that COP0 counts *every* instruction issued. The instrumentation calls ATtaCK inserts into the application get counted just like your original program—unlike ATtaCK, COP0 doesn’t know the difference. Likewise, if the CPU processes an exception or interrupt during your program, the COP0 counters just keep on ticking. Thus, most of the time you’re better off using ATtaCK instrumentation to measure instructions rather than using the performance counters—the speed hit is worth it to get results that are more representative of your actual application performance.



**Load/store:** Counter 0 can measure every time you load data from memory, while counter 1 can measure every time you store data into memory. Accessing memory counts as a load or store regardless of whether it came out of the cache or from actual system RAM. As with instructions, this is something you could very easily measure using ATtACK, at the cost of degrading the application's performance.

**None:** Either or both counters can be set to measure nothing at all, which is the default.

Okay, now that you understand how the performance counters work, the analysis code will be easy: All it does is activate, manage and read the performance counters. We'll take a look at that next.

## ***PS2Counter: Analysis Code***

The only way to work with the performance counters, or COP0 in general, is through special assembly instructions. That means we're going to have to look at some assembly code. It's really *simple* assembly code, though—particularly compared to some of the tricks one can pull with the MIPS instruction set! In fact, this code is so simple that you can basically just imagine the assembly instructions as being API function calls. You'll see what I mean about that in a minute.

Open the PS2Counter example program's analysis-code project—the full path to this should be `C:\Program Files\Metrowerks\ATtACK for PS2\Examples\PS2Counter\Anal\ps2counter_anal.mcp` for a default installation. Now open `ps2counter_anal.c`.

The top of the file is the usual boilerplate. There's a single data buffer, to which the code maintains a pointer. This buffer contains one `procdat_t` structure per procedure, storing the number of instructions executed and accumulating the performance-counter events.

`HostEnable` is a global flag that allows the host to turn gathering on and off: When you hit ENTER on the host to start collecting data, the host halts the target and writes `TRUE` into this memory location. (Technically, this variable ought to be marked `volatile`, since its value is expected to change from outside the program, but we get away with it here.)

The constant value `NFRAMES_SETTLE`, the countdown variable `CountDown` and the flag `Collect` are used to delay the start of collecting data. To enable data gathering, the host has to halt the target, write to its memory and then resume it, operations that consume time and system resources. Moreover, when the target restarts, the instruction and data caches will have been disrupted, disabled interrupts might need to get handled and so forth. The analysis code thus counts down ten frames, then sets `Collect` to `ATCK_TRUE` to start collecting data, giving the system time to “settle” back into its normal state.

## Initialization Routines

Well, okay, there's only one function here, but saying "initialization routine" didn't sound right. The `Initialize()` function first stores the received pointer to the buffer `ATtaCK` allocated. It then passes a magic number to the `mtps()` function, which is used to activate and configure the event counters. Truth be told, I don't know how this magic number is generated, so let's just take it as gospel.

## Analysis Routines

`Enable()` is called at the start of every frame. Mostly it does nothing. If, however, the host has enabled data collection, then `Enable()` decrements the countdown variable. When that variable reaches zero, the function turns on the performance counters and starts gathering data.

`Disable()` is called at the end of every frame. If `Collect` is true, then this function turns data collection back off. It then halts the target, allowing the host to read the gathered information.

`EnterReset()` is called at the start of every procedure. If data is being collected, this function increments the procedure's instruction counter, then uses the `mtpc` assembly instruction, "Move To Performance Counter," to clear both of the performance counters. The syntax for this instruction is very simple:

```
mtpc src, dest
```

*src* is the EE core register to copy into the counter, while *dest* specifies the counter (either 0 or 1—the dollar sign here simply indicates that this is a number). Note, this violates the normal "dest, src" convention. Also note that the EE core registers are 64 bits wide while the performance counters are only 32 bits wide, so the upper 32 bits of the core register are discarded.

In this case, we copy register `GPR0` into the two counters. This special register always contains the value 0, so in effect we're resetting the counters.

`Reset()` works just like `EnterReset()`, except that it's called within a procedure to reset the counters as necessary. As such, it doesn't increment the instruction counter, since that's already been done at the top of the procedure. `Reset()` is generally used after returning from procedure calls, to keep the called procedure's events from "contaminating" the caller's profile.

## Helper Routines

The next two functions, `getcounter0()` and `getcounter1()`, are written entirely in assembly language—as well they should be, considering how trivial they are! Each function first issues a `jr ra` instruction, starting a return to the caller. Branches always execute the next instruction after the branch before transferring control, and in this case that next instruction is `mfpcc`. This instruction, "Move From Performance Counter," is just the reverse of `mtpc`, with the following syntax:

```
mfpcc dest, src
```

This time it *does* follow the normal convention: *dest* is the EE core register to copy into, while *src* specifies the counter (either 0 or 1 again). The 32-bit counter is sign-extended to fill the 64-bit destination register.

`Accumulate()` is a simple C function that uses `getcounter0()` and `getcounter1()` to read the performance counters and add them to the running totals stored in the profile data buffer.

The `mtps()` function is a wrapper for `mtps`, the “Move To Performance-event Specifier” instruction. This instruction has the same syntax as `mtpc:mtps src, dest`. For *src*, we use `GPR4` (also known as `a0`), the register holding the first function argument. Unlike the performance *counters*, there’s only one performance-event *specifier*, and so the only valid value for *dest* is 0.

The next instruction after `mtps` is `sync.p`. This simply tells the processor to wait until the previous instruction finishes execution. This is necessary because we don’t want to leave this routine without knowing that the performance counters are properly initialized.

The more assembly-aware of you might now be asking, “If `sync.p` is necessary after `mtps`, why isn’t it necessary after `mtpc`?” Good question! The answer is that it *is* necessary we just don’t care. Yes, it’s possible that by failing to sync after clearing the performance counters, we will miss a few processor events. However, given that we have to clear the counters at the start (and often end) of every function call, putting `sync.p` there would restart the pipeline and utterly kill our performance. On the other hand, you’ll want to go ahead and synchronize writing analysis tools that demand higher accuracy than this general-purpose one.

At the end of the function, we do the standard `jr ra` to return to the caller. The `nop` (“No OPeration”) instruction is just a placeholder—the delay slot has to have *something* in it, and `nop` is harmless. The only other thing we could put there is the `sync.p` instruction, which unfortunately is illegal in delay slots.

You can probably guess what the last two functions do. That’s right:

`enable_counters()` enables the counters, and `disable_counters()` disables them. This is done by setting or clearing bit 31 of the performance-event specifier register.

First, we use `mfps` to fetch the current value into `GPR1`. Then `lui` loads the immediate value `0x8000` into the topmost 16 bits of `GPR2`, clearing the bottom 16 bits. (The folderol is necessary because MIPS chips can’t load immediate 32-bit values.)

To enable the counters, `GPR2` is Ored against `GPR1` and the result stored in `GPR1`. To disable the counters, `GPR2` is negated and then ANDed into `GPR1`. In either case, we now use `mtps` to store `GPR1` into the performance-event specifier. A `sync.p` to make sure the change “takes,” the traditional `jr ra/nop` pair, and we’re all done!

Okay, you get the idea. The COP0 performance counters do all the *real* analysis work, of course—this code just sets them up and lets them do their job. Now let’s look at the instrumentation process.

## PS2Counter: Instrumentation

Open up the instrumentation tool project, which is probably `C:\Program Files\Metrowerks\ATtaCK for PS2\Examples\PS2Counter\Inst\ps2counter_inst.mcp`. Within this project open `ps2counter_inst.c`.

The `main()` function is largely boilerplate, which at this point in the course I think we can safely ignore. Let's jump straight to the good part, starting with the `Instrument()` function.

### Instrumentation

First, we declare the `PrDat` structure and the `Initialize()` analysis routine. Then we insert a call to `Initialize()` at the start of the target application. In addition to the profile data buffer, this routine gets passed flag values indicating which events to log in counter 0 and counter 1. (Note that the code here talks about “counter 1” and “counter 2,” but as these are more correctly termed 0 and 1, that's how I'll continue to refer to them.)

As we saw previously, the analysis routines are designed to gather a single frame of data at a time. Ideally, there is some function in the target application that is already called at the top of each frame—for example, there might be a function that handles all the world physics, which would generally be the first thing processed each frame of the game loop.

If there's some convenient function available to indicate the start of a frame, the user specifies it on the tool's command line, and the instrumentation code inserts a call to `Enable()` at the start of the function. If there's *not* such a function, then one will have to be created in order to use this tool. Likewise, the user must specify a function name to mark the end of a frame; a call to `Disable()` is inserted before that function.

With that out of the way, the instrumentation code now declares the remaining three analysis routines, and starts iterating over the application's procedures. This is where things really get interesting—open up `classify.c`.

### Classification

This file contains the function `Classify()` and its support functions. `Classify()` takes a procedure and determines which of three categories it belongs to:

**COUNT:** Procedures in this category get instrumented and profiled normally. This is the default. The procedures specified by the user as marking the start and end of each frame are *always* classified as `COUNT`, since the data-collection system relies on those routines getting instrumented.

**INPARENT:** Procedures in this category are profiled as part of their callers. In other words, the performance counters are not reset when these procedures are called, and the procedures aren't instrumented. For all intents and purposes, the tool treats these procedures as if they were inlined.

So what qualifies a procedure as `INPARENT`? Well first of all, if a procedure can't be instrumented—for instance, if it's part of the SCE libraries—then it's a “black box.” The only way to profile such procedures is via the performance counters. By clearing the counters before each call, and checking them after the return, we *could* profile these routines independently. However, what would we actually do with that information? It's not as though we can optimize the library code! So, we will simply treat these library routines as if they were part of their callers.

Procedures are also classified as `INPARENT` if they're “simple.” A simple procedure has no loops, makes no calls to non-simple procedures and is less than 100 instructions long. Such procedures don't offer a lot of opportunity for individual profiling and tuning, so we just treat them as part of the parent. Indeed, when performance is more critical than code size, such procedures are prime candidates for being inlined, at which point they really *are* part of the parent.

The test for simplicity is done in `IsSimple()`. This function simply iterates over the instructions within a procedure. If an instruction is a callsite, then the target procedure is itself tested for simplicity by a recursive call to `IsSimple()`. (Of course, this means that if you have two functions in your application that call each other, PS2Counter will crash with a stack fault. In the assignment at the end of this lesson, we'll look at one way to keep this from happening.)

If an instruction is a branch, then the function gets the target address of the branch and checks to see whether that address is greater than or less than the current instruction's address. Note that, while callsites are also branches, we've already checked whether the instruction is a callsite before we get here, so we know that the branch target is within this procedure. A branch backward to an earlier instruction within the same procedure is, almost by definition, a loop—it's possible to write assembly code in which that's not true, but the compiler doesn't do that.

**EXCLUDE:** Procedures in this final category are excluded from the profile entirely. These procedures, identified by name, are ones known not to offer any useful profiling information. Right now this consists only of the library routines `sceGsSyncV()` and `sceGsSyncPath()`. Normally, these routines would be classified as `INPARENT`. However, these routines' idle loops generate large numbers of system events (especially instruction issues and instruction-cache misses) that will obscure the relevant profiling results.

Feel free to add any other routines to the exclusion list that you see fit. Remember the principle advocated earlier in this lesson: Don't collect information that you're not going to use.

Now go back to `ps2counter_inst.c`, where we'll wrap up navigation and look at execution.

# ***PS2Counter: Navigation, Execution and Output***

## **Navigation and Instrumentation**

The tool now iterates over every procedure in the application, classifying it. The instrumentation depends on the procedure's category:

### **COUNT**

This category is the default. Procedures in this category get instrumented first with a call to `EnterReset()` at the start, and `Accum()` at the end. Clearing the performance counters at the start and reading them at the end collects profile data for just the single procedure.

Once those calls are in place, the navigation code looks inside the procedure for call sites. Calls to `COUNTED` and `EXCLUDED` procedures are not counted inside this procedure—`COUNTED` procedures are profiled in their own right, while `EXCLUDED` procedures aren't profiled at all. Thus, these calls are instrumented with `Accum()` beforehand and `Reset()` afterwards.

Calls to `INPARENT` require no special handling, since they're supposed to be part of the caller's profile.

### **INPARENT and EXCLUDE**

If a procedure is `INPARENT`, then its profile is included in that of the parent, and so we don't need to instrument it. If a procedure is `EXCLUDE`, then we don't *want* to instrument it. They amount to the same thing: Procedures in these categories are simply skipped.

Note that indirect calls—for example, C++ virtual function calls—can't be classified, since the target is unknown at instrumentation-time. These default to `COUNT`, isolating them from the caller's profile to make sure they don't get counted twice. However, if these procedures are in fact simple, then when it's their turn to be instrumented, they'll be classified as `INPARENT` and won't get counted at all. The upshot of all this is that PS2Counter is really not designed for C++, so if you're working in C++ you'll need to customize the tool.

## **Download, Execution and Output**

Once the instrumentation is finished, the tool calls `RunIt()`. This function is mostly boilerplate, connecting to the target and launching the application immediately. The function then drops into a loop waiting for user input. When the user hits `ENTER`, the loop calls `SnapshotFrame()`.

`SnapshotFrame()` first halts the target and sets the `HostEnable` flag to `TRUE`, then resumes the target. When the `Enable()` analysis routine gets called at the start of the next frame, the target realizes that `HostEnable` has been modified—that's the only way a target application can detect that it's been suspended by the host. This starts the “settle” countdown process, at the end of which a single frame of profile information is gathered. Once that frame ends, the target halts itself.

Most ATtaCK tools read their information from the target in an event handler, since they can't predict when the target application will stop. In this case, though, the tool knows that the target will stop very soon (1/6<sup>th</sup> of a second, generally) after the user hits ENTER. Thus, the tool simply calls `atck_wait()` to suspend itself until the target stops.

When `atck_wait()` returns, we know that the target has a frame of data ready to read. The `DumpCounters()` function handles this, in a wholly unremarkable way. Its last act is to zero-out the profile data buffer on the target—this is purely a one-frame-at-a-time profiler.

```
D:\sbtest\PS2Counter\bin\ps2>ps2counter D:\CodeWarrior\Examples\sce200\vu1\blow\
blow.elf imiss dmiss FrameBegins FrameEnds
Downloading... 18%
Downloading... 36%
Downloading... 55%
Downloading... 73%
Downloading... 92%
Downloading... 100%
Press ENTER to dump counter output, x-ENTER to quit.

Procedure           Calls      I$ Misses      D$ Misses
-----
FrameBegins         1           0             0
FrameEnds           1           1             1
main                0           61            120
CreateViewingMatrix 1           91             61
SetUViewPosition    1           11             2
SetParticlePosition 1           25            4694
-----
Totals              5           189           4878
x
D:\sbtest\PS2Counter\bin\ps2>
```

*Fig.*

*05-01: PS2Counter Output*

## Whew!

And with that, we're now done with the entire API—at this point you should be able to create any ATtaCK tool you need. Our last two lessons are labs, in which we'll look at eight more “real-world” tools to give you more. Those should go pretty quick, so let's go ahead and get started with Lesson 06: Profiling Applications.

## Lesson 05 Assignment

Tasked with fine-tuning your PlayStation 2 game's render system, you break out PS2Counter to get some initial, overall information. However, when PS2Counter tries to instrument your application, the tool crashes with a stack fault.

Fortunately, you remember reading in this course that PS2Counter cannot handle recursive functions. A quick check with the other programmers reveals that, sure enough, the particle-animation system uses a recursive function. But you still need to profile the application.

Fix PS2Counter so that it no longer crashes when trying to instrument recursive functions. The crash happens in `IsSimple()`, found in `classify.c`, and can be prevented just by modifying that function.

### Hint:

Remember from Lesson 02 that two handles to the same object will always be the same, and can be tested for equality using `==`.

### Answer:

There's basically two ways to fix this problem. To reinforce the hacker mindset you should use with ATtACK tools, I'm only going to present the "fast and dirty" solution. For the purists, though, I'll tell you the "proper" solution at the end.

The basic problem is that, if a function being analyzed by `IsSimple()` calls itself, then it will call `IsSimple()` on itself, which in turn calls `IsSimple()` on itself, and so forth, and so forth...

So your goal is to keep `IsSimple()` from calling `IsSimple()` again on the current function. That turns out to be very easy: You already know the handle to the current function (it's the argument `pproc`), and you know that, no matter what, every procedure handle that refers to that same function will have the same value. Thus, you can just check to see whether the target procedure's handle, as returned from `atck_ent_proc()`, is equal to `pproc`—if it is, then *don't* call `IsSimple()`! In fact, since a function that recursively calls itself is pretty much not simple by definition, you can just return `ATCK_FALSE` without further ado.

So the quick-and-dirty fix is to turn this (line 146 or thereabouts):

```
if (!pent || !IsSimple(atck_ent_proc(pent), &ninst2)) {
```

into this:

```
if (!pent || atck_ent_proc(pent) == pproc ||  
    !IsSimple(atck_ent_proc(pent), &ninst2)) {
```

Notice that, since the compiler endeavors to resolve the if statement as quickly as possible, the second clause won't get executed if the first is true, and the



third won't get executed if the second is true. Therefore, if the target of the callsite in question is this function itself, `IsSimple()` returns `ATCK_FALSE` immediately rather than getting caught in an infinite loop.

I promised the purists the “proper” solution, which requires stating what the real problem is: If X calls Y, and Y calls Z, and Z calls X, then when `IsSimple()` tries to analyze X, it will inevitably drop into the same infinite loop as if X called X. Our quick-and-dirty fix does nothing to prevent this general-case bug.

What we'd have to do to *really* fix the problem is maintain a stack of parent procedure handles. Every time `IsSimple()` is entered, it pushes the `pproc` argument onto that stack; every time it exits it pops the stack. Every time `IsSimple()` is about to recurse into itself, it searches the stack to make sure that the target procedure isn't a parent of the current procedure.

Now you can see why I focused on the quick fix. The “right” solution is big and complicated, and unnecessary in the most common case (X calling X). In between projects, when you're honing your analysis tools, you can implement this sort of general-purpose fix, to save yourself time down the road. But in the middle of a project, go for the quick hack. You're here to fix your game, not your tools.

## ***Lesson 05 Quiz***

1. True or false: The only purpose of analysis tools is optimization.
  - A. True
  - B. False
2. True or false: Changing your algorithm is often the best way to fix not only performance problems but bugs of all kinds.
  - A. True
  - B. False
3. Which of the following is always feature of a well-designed ATtaCK tool?
  - A. Flexibility
  - B. Good programming style
  - C. Simplicity
  - D. A graphical user interface
4. True or false: The best place to analyze your data is on the target, in the analysis routines.
  - A. True
  - B. False
5. True or false: It's easiest and safest for analysis routines to read registers directly, rather than relying on ATtaCK to pass them in.
  - A. True
  - B. False
6. Which of the following is a valid ATtaCK analysis-routine declaration on the PlayStation 2?
  - A. `"GetTarget(valaddr)"`
  - B. `"GetTarget(ATCK_TARGADDR)"`
  - C. `"GetStack(ATCKMIPS_REG_GPR29)"`
  - D. `"GetStack(valuint64)"`
7. Which of the following is a valid place to use `ATCK_EFFADDR`?
  - A. `atck_proc_callafter()`
  - B. `atck_call_callbefore()`
  - C. `atck_inst_callbefore()`
  - D. `atck_inst_callafter()`

8. True or false: You must always use `atck_inst_isallowed()` to verify that a dynamic argument is safe before trying to instrument an instruction with it.

- A. True
- B. False

9. True or false: An address is all that's required to completely identify any instruction in a program.

- A. True
- B. False

10. Which of the following does a MIPS application (that is, a PlayStation 2 game) store on the stack?

- A. Function arguments
- B. Local variables
- C. Return addresses
- D. None of the above

## Lesson 06: Profiling Applications

---

The first five lessons cover everything you need to know to develop ATtACK tools. Our sixth lesson will really be more of a lab, as we examine four tools to tackle common profiling and debugging tasks: verifying code compliance, examining your branch-prediction performance, calculating which functions are worth inlining and catching registers that are used before they finish loading from memory. These tools will probably be useful to you as is, but we'll also spend plenty of time discussing ways to improve and tailor them to your exact needs.

### ***Verifying Code Compliance***

Code validation is a form of static analysis in which you review your application looking for certain types of expected problems. For example, when you finish writing a function, you might go back and review it to make sure you initialized all its variables. That's a rudimentary form of code validation—the “expected problem” you're looking for is an uninitialized variable.

The best-known example of code validation is lint, a tool that reviews C source for common (and not-so-common) errors. Uninitialized variables are just one of many, many problems that lint will catch. The value of code validation is that these problems might not show up during normal testing and debugging.

ATtACK can be used for binary code validation, looking for expected problems in the binary code. The last program we'll look at in this lesson, for instance, scans the binary code looking for places where you use a register immediately after it's loaded from memory.

The first tool we'll be examining also does code validation. Called TRCTool, it tests for Sony TRC compliance. As part of the PlayStation 2 licensing approval process, Sony prohibits certain functions from being called in shipping applications. These are kernel functions which are valid and useful for debugging, but which could crash an end-user's system. There are other functions that Sony *requires* be present, various initialization library routines that must be called for a PlayStation 2 application to function properly.

TRCTool checks for both these categories of functions. It reads a list of them from a text file, and then scans the program to see if any prohibited functions are called, and conversely to make sure that all the required functions are called.

Now, this tool isn't foolproof. TRCTool uses static analysis only, which means that it's basically predicting what the program will do in action. Since that prediction is imperfect, the tool can be fooled. For instance, if your application calls a required function conditionally, the condition must be true for the function to be executed. TRCTool doesn't try to check whether the condition is always (or ever!) true, it just notes that the required function call is present. The tool is really just for pre-screening your application, so static analysis like this is sufficient.

## TRCTool

Open the project file, `trc_tool_ps2.mcp`, located in the Thrill Seeker Tools\TRC\inst subdirectory off your ATtaCK folder. (On a default installation, this will be `C:\Program Files\Metrowerks\ATtaCK for PS2\Thrill Seeker Tools\TRC\Inst\trc_tool_ps2.mcp`.)

Right away you'll notice something different—this program's written in C++! As you'll see next page, the tool spends most of its time just parsing its configuration file, a task that is easier to manage in an object-oriented language.

Normally I wouldn't recommend writing an ATtaCK tool in C++—the “quick-and-dirty” coding approach ATtaCK facilitates doesn't mix well with object-oriented programming. However, if you're writing a tool that you intend to extend and reuse over several projects, the maintenance and design burden of writing good C++ code can pay for itself.

(Later on in this lesson, we'll see a quick-and-dirty tool written in C++ for another reason: to take advantage of the STL.)

Now that you're past the shock of seeing `.cpp` file extensions, let's review the project's contents:

- `trc_tool.cpp` contains `main()` and does all the analysis and reporting.
- `ProcedureCallDetails.cpp` and `.h` implement the `ProcedureCallDetails` object, used for holding the analysis results.
- `TRCConfiguration.cpp` and `.h` implement the `TRCConfiguration` object, which parses and contains the lists of prohibited and required functions.
- `TRCProcedure.cpp` and `.h` implement `TRCProcedure`, a helper object used by `TRCConfiguration`.
- `CallSite.cpp` and `.h` contain `CallSite`, a C++ wrapper object for `atck_call_t`.
- `UnderterminedCallSite.cpp` and `.h` contain the `UndeterminedCallSite` object, another C++ wrapper, and this one for sites that make indirect function calls—i.e., whose targets cannot be determined. In spite of its name, it doesn't inherit from `CallSite`.

Strangely enough, the most complex code in this program is found in initialization, so we'll start there.

## ***TRC: Initialization and Navigation***

### **TRCConfiguration Object**

It's really a testament to the ease of writing ATtaCK tools that the most complex code in this tool—nearly the most complex code in this entire course!—is the code to read and parse the configuration file. Actually, ATtaCK provides code to

do *this* too, but this particular program doesn't take advantage of that. The good news is that, this being C++, you can easily take this parsing code and adapt it to your own projects.

The `TRCConfiguration` object has a fairly standard interface that doesn't call for a lot of examination. The object maintains three lists: prohibited functions, required functions and SCE library functions. The first two lists have been discussed. The third list is used to allow the tool to identify and skip any functions provided by the SCE libraries.

The prohibited and required lists are composed of `TRCProcedure` objects, which are just structures containing two strings, the name of the procedure and the reason why it's either required or prohibited. These lists are public members, implemented using the `vector` STL class. By contrast, the "SCE library routines" list is protected, its implementation hidden from the user by an access method, `bIsSceFunction()`. That's definitely better design, but either way works.

The other access methods are `strGetVersion()` and `strGetConfigFilename()`. The first returns a version string read from the configuration file, while the second just returns the name of the configuration file itself.

That's about it for the object's interface—pretty minimalist. All the work happens in the initialization method `vParseConfigurationFile()`.

## Parsing

`TRCTool` expects to find four members in its configuration files: `VERSION`, a version string; `SCE_FUNCTIONS`, the name of a text file to parse for a list of SCE functions; `REQUIRED`, the list of required functions; and `PROHIBITED`, the list of prohibited functions. For an example file, open up `trcl_6.txt`, found in the same folder as the project.

Each of these members is represented using a simple XML-like syntax. The two lists are delimited with open and close tags (for example, `<REQUIRED>` and `</REQUIRED>`), while the two string members are contained in self-closed tags (`<SCE_FUNCTIONS file="sce_functions.txt"/>`). Any line beginning with `#` is a comment, ignored entirely.

`vParseConfigurationFile()` uses a simple state machine to parse this format. The self-closed tags don't need states, since they're parsed entirely when encountered. The opening and closing tags for the prohibited and required lists trigger state transitions; within each state, every line read is passed to `vParseFunctionSpec()`.

`vParseFunctionSpec()` receives a reference to the list being read, and a line of text to parse. The first thing expected on each line is the procedure name, which of course must not have any spaces. After a space comes the reason why this procedure belongs to the list—just a simple text string, surrounded by quotes.

Unlike the other two lists, the SCE functions list is read from another text file. The name of that file is specified in the `SCE_FUNCTIONS` tag. The file itself, parsed by `vLoadSceFunctions()`, just contains one name per line, with no comments.

This file is created automatically from the SCE library symbol table, using a Perl tool contained in the `TRC\GrabSCESyms` folder. Since this isn't a Perl course, we'll ignore that tool—whenever you're curious, it'll be there waiting for you.

## Navigation

With that out of the way, go back to `trc_tool.cpp`. Once the configuration file is read, we get back to the standard boilerplate that's becoming so familiar. After that, we clear our two lists of call sites. We'll see how those get used in a moment.

Unlike the other ATtaCK tools we've seen, this one uses recursion rather than iteration to walk the program. In iteration, a tool runs through every image, then through every procedure in each image, and then through every call site in each procedure. In recursion, though, the tool takes a particular procedure and runs through every call site in it; for each call site, the tool finds the target procedure and runs through every call site in *it*.

This process starts in `vForeachCallSiteFromMain()`, so that's where we'll start too.

### **vForeachCallSiteFromMain()**

This function takes two arguments. The first is the handle of the program to analyze. The second is a pointer to a function to invoke on each call site in the program's chain of execution. Ignore that for now—we'll look at it in detail soon enough.

The first thing this function does is check to make sure the program is instrumentable. This is still a static analysis tool, however, so it *can* run against non-instrumentable programs. The most likely reason for a program not to be instrumentable is that it was compiled with `gcc` rather than `CodeWarrior`. By default, the tool will run against `gcc`-compiled applications, but if for some reason you don't want it to, you can pass in the command line option `-no-gcc` to prevent it.

Next, we look up the entry point for `main()` and pass it to `vRecurseCallSites()`, the function that will do the recursion.

### **vRecurseCallSites()**

This function takes two arguments. The first is the handle of an entry point. The second is the function pointer we've already mentioned. Keep ignoring that function pointer and just look at what we do with the entry-point handle.

The function looks up the procedure handle associated with the entry point, and then gets that procedure's instrumentability attribute. Instrumentable procedures get analyzed, as do uninstrumentable ones in the absence of the `-no-gcc` option. However, procedures from the Sony libraries—that is, those for which `bIsSceFunction()` returns true—are skipped.

If the procedure isn't skipped, we proceed to iterate through all its call sites. If the target of a call site is known, then we call our static-analysis code—the function to which we received the pointer—on that target, and then pass the target entry point into `vRecurseCallSites()` itself. If, on the other hand, the site's target is unknown, we call `vRecordUndeterminedCallSite()` instead.

Now let's look at our static-analysis routines. Go on to the next page.

## ***TRC: Analysis and Reporting***

### **Analysis**

The recursion routines call a function on every call site in the program. That function is identified by a pointer for versatility: You can use the same routine to invoke any function recursively over the program. In this tool, though, only one function gets called, `vAddCallSiteToHashtable()`.

#### **vAddCallSiteToHashtable()**

This function creates a new `CallSite` object, which represents a single call site in the program. This object, along with the name of the target procedure, are passed to `addProcedureCallToHashtable()`. (Yes, that's `Hastable`, not `Hashtable`—the source contains a typo. Now you know why this is in the “Thrill Seeker Tools” folder.)

To add the call to the table (which is really implemented using a vector, not a hashtable—the hashtable is on the “to-do” list), first the routine scans through the table to see whether a `ProcedureCallDetails` object with that name already exists. If it does, then the callsite is added to the object's internal list. Otherwise, a new `ProcedureCallDetails` object is created.

### **CallSite Object**

This is just a wrapper for ATtaCK's `atk_call_t` object—it's simple, but it gives you an idea of how a well-designed set of C++ wrapper objects could improve ATtaCK development. Its constructor takes handles of the procedure containing the call, the ATtaCK callsite object, and a boolean flag indicating whether the site appears in read-only code. Access methods are provided for these members, plus there's a fourth access method that wraps `atk_call_targname()`.

### **ProcedureCallDetails Object**

This object has just two members: the name of a target procedure, and a set of the `CallSite` objects, which target that procedure. The object also has methods that provide access to those members—pretty basic stuff.



## vRecordUndeterminedCallSite()

When the target of a call site can't be determined, this function sees whether the site is already stored in the list of undetermined call sites. If it is, then no further action is needed. If not, a new `UndeterminedCallSite` object is created and added to the list.

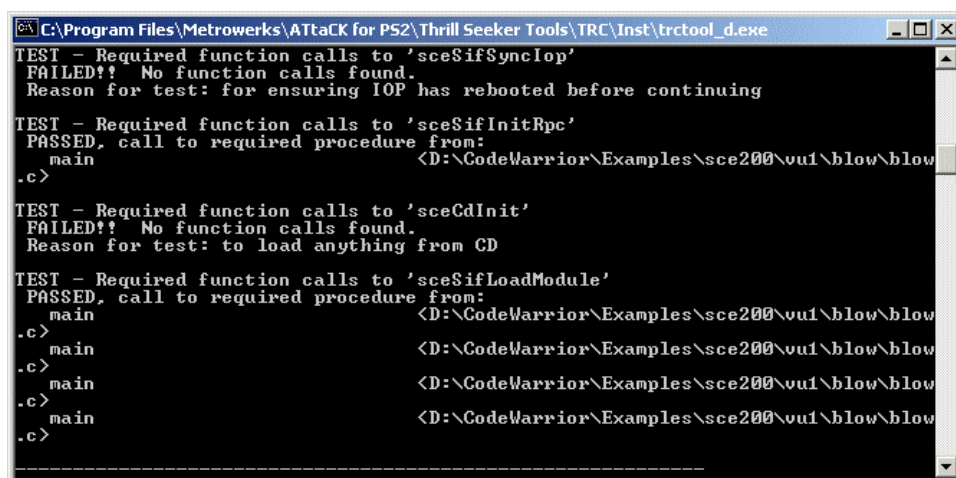
## UndeterminedCallSite Object

Indirect call sites might or might not be calling prohibited or required functions. Since the tool can't figure them out, it simply reports the call site to the user. The `UndeterminedCallSite` object encapsulates the little information the tool *does* have about the site—primarily the source filename and line number where the site can be found.

## Output

The tool's syntax is very simple: `trctool configfile app.configfile` is the configuration file—use `trcl_6.txt`, located in the same folder as the tool. `app` is the application to validate.

Once launched, the program runs for a little while and spits out its results, looking something like this:

A screenshot of a Windows command window titled "C:\Program Files\Metrowerks\ATtack for PS2\Thrill Seeker Tools\TRC\Inst\trctool\_d.exe". The window displays the output of the TRCTool application. The output consists of several test results for different functions. The first test for 'sceSifSyncIop' failed because no function calls were found. The second test for 'sceSifInitRpc' passed, showing a call from 'main' in a file at 'D:\CodeWarrior\Examples\sce200\vu1\blow\blow.c'. The third test for 'sceCdInit' failed because no function calls were found. The fourth test for 'sceSifLoadModule' passed, showing multiple calls from 'main' in the same file. The output is formatted with 'TEST - Required function calls to', 'FAILED!! No function calls found.', 'Reason for test:', 'PASSED, call to required procedure from:', and the file path and line number for each call.

```
C:\Program Files\Metrowerks\ATtack for PS2\Thrill Seeker Tools\TRC\Inst\trctool_d.exe
TEST - Required function calls to 'sceSifSyncIop'
FAILED!! No function calls found.
Reason for test: for ensuring IOP has rebooted before continuing

TEST - Required function calls to 'sceSifInitRpc'
PASSED, call to required procedure from:
    main                                <D:\CodeWarrior\Examples\sce200\vu1\blow\blow
.c>

TEST - Required function calls to 'sceCdInit'
FAILED!! No function calls found.
Reason for test: to load anything from CD

TEST - Required function calls to 'sceSifLoadModule'
PASSED, call to required procedure from:
    main                                <D:\CodeWarrior\Examples\sce200\vu1\blow\blow
.c>
    main                                <D:\CodeWarrior\Examples\sce200\vu1\blow\blow
.c>
    main                                <D:\CodeWarrior\Examples\sce200\vu1\blow\blow
.c>
    main                                <D:\CodeWarrior\Examples\sce200\vu1\blow\blow
.c>
```

Fig. 06-01: TRCTool Output

## Improvements

TRCTool isn't ready for prime time yet. It's got three notable flaws: It can't handle recursive applications; it doesn't guarantee that the required functions are actually called; and it can't handle indirect function calls.

Of these, the recursion problem is unquestionably the worst. Right now, the tool doesn't have any mechanism to indicate that a routine has already been examined. For most applications, that's merely wasteful. If the target application itself uses recursion, however, TRCTool will crash. Recursive algorithms are not at all uncommon in games, especially in AI routines and particle-animation systems.

Fortunately, there's an easy fix available. Before recursing into a call-site's target, the tool calls `vAddCallSiteToHashtable()` for it. That function in turn calls `addProcedureCallToHashtable()`. And *that* function determines whether the procedure already exists in the table—in other words, that function knows whether the procedure has been examined already.

Thus, all we need to do to fix this bug is pass a boolean return value back from `addProcedureCallToHashtable()`, then pass that same value back from `vAddCallSiteToHashtable()`. This value should be `false` if the procedure was already in the table, or `true` otherwise. We can then make the call to `vRecurseCallSites()` conditional on this return value.

The other two problems are not nearly as severe, but they're not nearly as easy to fix, either. Guaranteeing that a particular function gets called requires dynamic analysis, adding instrumentation to every call site that targets required functions. The instrumentation should pass a "required function ID," which the analysis code can use to check the call off a list.

With that mechanism in place, handling indirect function calls becomes a simple matter. Add instrumentation to every indirect call that passes the called address via `ATCK_TARGADDR`. The analysis code's required-function table would then have an address listed for each function. The called address would be matched against the known addresses of the required functions to see whether one should be checked off. Since there are only a handful of required functions, this wouldn't slow the program down noticeably.

## ***Analyzing Branch Prediction***

Because the EE is so heavily pipelined, branches present a special problem. By the time the processor determines whether the branch will be taken or not, it already has as many as six instructions *after* that branch loaded in the pipeline. If the branch isn't taken, those instructions execute normally and no time is lost. If the branch is taken, however, those instructions must be flushed out of the pipeline, which starts all over again at the target.

This can waste ten nanoseconds per branch, which may not sound like much but it adds up. Worse yet, if any of those six instructions lie in another cache line—which, since cache lines are 16 instructions wide, they will about 37% of the time—then you incur the hit of a wasted cache miss in addition to the cache miss caused by the branch itself. (If you're unfamiliar with terms like "cache miss," just wait—we'll be covering the cache in detail next lesson.)

The EE instruction set gives you two ways to alleviate this problem. First, the Jump instructions provide for unconditional branches. Since the chip knows that the branch will be taken, it doesn't bother loading the instructions beyond it.

Second, the Branch-Likely instructions allow you to identify conditional branches that are usually taken. As with Jumps, Branch Likely instructions assume that the branch will be taken, and start loading the pipeline with instructions from the

target address. Of course, if a Branch Likely turns out to fall through instead, then the hit is just as bad as when a regular conditional branch is taken.

Now, branch prediction isn't nearly as important as managing the cache well. Nevertheless, it's still an important speed factor. It's also a factor that's relatively easy to tweak. In assembly, you can replace Branches with Branch Likelies. In C, you can reverse the sense of `if` statements. We'll discuss both those optimizations later in this lesson.

But the first step in solving a problem is recognizing you have one. You need a tool to monitor your application's branches, logging the missed predictions. By an amazing coincidence, such a tool is already sitting on your hard drive. Open up `branch_inst_ps2.mcp` from your `Examples\Branch\Inst` folder (by default, this will be `C:\Program Files\Metrowerks\ATtACK for PS2\Examples\Branch\Inst\branch_inst_ps2.mcp`.)

## Branch

This is a very basic tool. It instruments every conditional branch with a call to a counter routine. ATtACK passes this routine a flag to indicate whether the branch is taken. This routine updates a counter for every branch, and a counter for every taken branch, to calculate the percentage of branches taken.

We can cover everything of interest here in about five minutes. Open `branch_inst.c`. The `main()` function is just a boilerplate that calls `Instrument()` and then `RunIt()`, so we'll skip that part.

`Instrument()` does two things. First, if the user specifies a procedure name on the command line, that procedure gets instrumented with a call to the analysis routine `Report()`. That routine halts the target, allowing the host to read the data. The net result of this is that the user can specify a procedure in the target application that causes the counters to be read. This function might be called at the end of each level, say, or upon pressing a control-pad button.

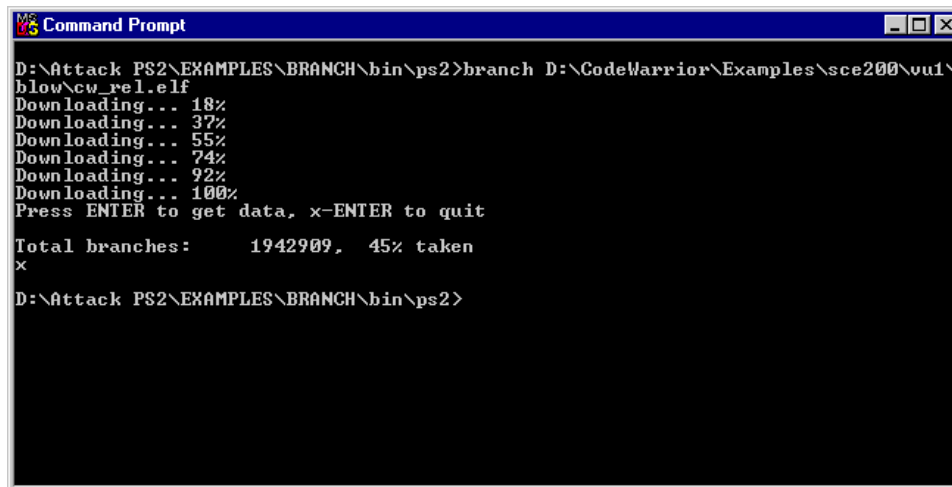
The rest of the application gets iterated over, instrumenting every instrumentable instruction that is a conditional branch. Conditional branches are identified using the `atck_inst_isbranch()` method of the instruction object. The instrumentation call passes the `ATCK_TAKEN` flag, which is received as `ATCK_FALSE` if the branch condition is false, or `ATCK_TRUE` if it's true.

The analysis code isn't even worth looking at, although it's in `Branch\Anal\branch_anal.c` if you're curious. It maintains two integer counters, one that tallies the total number of calls to the analysis routine—that is, the number of conditional branches—and one that counts the number of branches taken.

## Execution and Output

The syntax is `branch app report_proc`. The first argument is the application name, while the second is the name of the procedure to halt on—just leave that blank for now. The tool downloads and launches the application immediately; hit

ENTER to get the data, then x-ENTER to quit. Your output will look something like this:



```
D:\Attack PS2\EXAMPLES\BRANCH\bin\ps2>branch D:\CodeWarrior\Examples\sce200\vu1\
blow\cw_rel.elf
Downloading... 18%
Downloading... 37%
Downloading... 55%
Downloading... 74%
Downloading... 92%
Downloading... 100%
Press ENTER to get data, x-ENTER to quit

Total branches:      1942909, 45% taken
x
D:\Attack PS2\EXAMPLES\BRANCH\bin\ps2>
```

**Fig. 06-02: Branch Output**

As you can see, this just reports the total number of conditional branches, and the percentage that were taken. Don't you feel enlightened?

The trouble is, this tool violates one of my analysis principles: Don't ask questions if you're not going to use the answer. In this case, just what exactly are you going to do with the information that 45% of your conditional branches are taken? You don't even know whether that's good or bad!

If you think back to PS2Counter from the last lesson, you'll realize that this tool doesn't do anything the EE performance counters can't do. In fact, the performance counters do it better, since they measure every instruction, instrumentable or not, and don't slow down the application's performance at all.

But this tool's real sin is that it treats all conditional branches the same. Some of those conditional branches are Branch Likely, in which case we *want* them taken. Furthermore, we really need the count for each individual branch, so that we know which ones to tweak and which to leave alone.

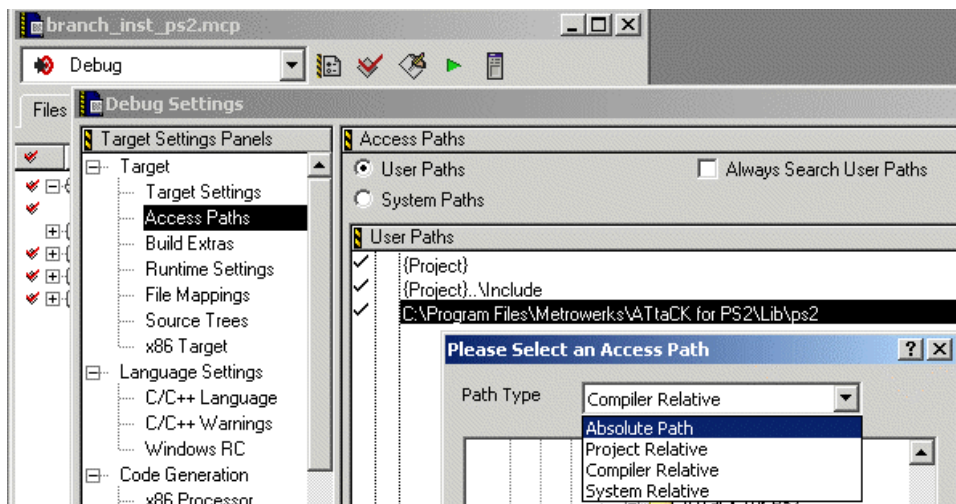
The whole point of ATtACK, though, is that you can write and modify tools as needed. So let's do exactly that and turn this program into something really useful.

## ***Branch: Improvement***

Despite its shortcomings, Branch is still a sample program, and we don't want to destroy it. But the easiest way to create our new tool, BranchPred, is to modify Branch. Besides, this is a good opportunity to give you experience in modifying existing ATtACK tools.

The first thing to do is change the Access Paths settings in the original Branch project from "Project Relative" to "Absolute Path." This won't break the original

project, and it will make sure that the new project compiles properly wherever we put it. Be sure to change the library path (found under “User Paths”) and the include path (found under “System Paths”) in both the instrumentation-tool and analysis-code projects, as show below:



**Fig. 06-03: Change Access Paths**

Once you finish changing the access paths, close each project to save the new settings. Now copy everything in `Examples\Branch` to a new folder elsewhere on your drive, named `BranchPred`. Don’t bother renaming any other files, although you probably do want to go in to the “x86 Target” settings and change the output filename from `branch.exe` to `branchpred.exe`.

Our basic goal is to add the ability to measure each branch’s prediction individually. While we’re at it, we’ll fix how Branch Likely instructions are handled, so that they count as missed predictions when *not* taken. This will require a number of small changes to both the instrumentation tool and analysis code, but nothing very big.

For making these changes, you’ve got two choices. The hard way is to figure out how to make them yourself, based on my step-by-step description. The easy way is simply to download the “new and improved” versions of the C files from the course supplemental material file—just save these new files on top of the old ones you copied from the original Branch folder.

## Navigation and Instrumentation

Open `branch_inst.c`, so that you can either make the following changes yourself (yay!) or see the final results (boo!):

1. Add `<atckps2.h>` to the list of headers. This file contains the PlayStation 2-specific definitions, which we’re going to need.
2. We’ll need a global pointer to a buffer of `brdat_t` structures, to hold the individual branch statistics. We’ll also need a global counter to hold the number of instrumented branches (and thus the size of the buffer). Call these `pBuffer` and `nBranches`.

3. We need a counter—call it `branchID`—declared at the start of `Instrument()`. This counter gets passed in as a new argument for the `TraceBr()` analysis routine. It's then incremented after every branch that's instrumented.
4. The analysis routine will need to be told whether the branch is likely or unlikely, so that it knows whether to log a taken branch as a hit or a miss. This will be done by a new function called `IsBranchLikely()`, which bases its decision on whether the branch instruction's "pseudo-opcode" matches a list of known Branch Likely opcodes.
5. In the analysis code, we'll add an "is branch likely" flag as an extra argument to `TraceBr()`; in the instrumentation loop, we pass the return from `IsBranchLikely()`.
6. After the instrumentation loop finishes, and we know how many branches there are, set `nBranches` equal to `branchID`. Then allocate a buffer of `nBranches brdat_t` structures, storing the pointer in `pBuffer`.
7. Finally, add an instrumentation call to a new analysis routine, `Initialize()`, at the start of the program. This routine takes a pointer to the buffer, which means that we must declare the `BrDat` structure, then pass ATtACK the number of buffer elements (`nBranches`) and the buffer address (`pBuffer`).

## Analysis Changes

We're not done with the instrumentation tool yet, but this is the logical time to look at the changes to the analysis code. Open `branch_anal_ps2.mcp`, then open `branch_anal.c` to make the following additions:

8. `BranchData` becomes a pointer to a `brdat_t` structure, rather than a `brdat_t` structure.
9. `TraceBr()` now takes three arguments: `branchID`, `predicted` and `actual`. You saw in the instrumentation call what those arguments are.
10. `BranchData.nbranches` becomes `BranchData[branchID].nbranches`.
11. `BranchData.ntaken` becomes `BranchData[branchID].ntaken`, and is only incremented if `predicted` equals `actual`.
12. We need a simple `Initialize()` function, which initializes `BranchData` with the `brdat_t` pointer passed to it.

Close `branch_anal.c` and make the analysis-code project. Now all that's left are the changes to the output code, which we'll tackle on the next page.

## ***BranchPred: Analysis***

### **Output Changes**

The only remaining function to change is `PrintData()`, as follows:

13. Before, the target's `BranchData` variable was a structure. Now it's a pointer to an array of structures. So now we need to read the contents of the pointer, then use that address to read the contents of the buffer.
14. Copy the entire iteration loop from `Instrument()` here, so that we can turn branch IDs into source filenames and line numbers. Don't forget to copy the declaration of the `branchID` counter, too!
15. Change the call to `atk_img_write()` at the end of the image iteration loop to `atk_img_release()`.
16. Replace the call to `atk_inst_callbefore()` inside the iteration loop with a call to the new function `AnalyzeBranch()`.
17. At the end of `PrintData()`, zero out the results buffer and write it back to the target.

Last, we need to create the `AnalyzeBranch()` function that reports the results of a single branch. It receives the branch ID and a handle to the instruction in question, and performs the following tasks:

18. Return immediately if the branch was never reached (i.e., if `pBuffer[branchID].nbranches` equals zero).
19. Calculate the percentage of time the branch was predicted correctly. If that value is greater than or equal to 50%, the branch is fine and we don't need to report anything about it.
20. If that value is less than 50%, however, report the source filename and line number of the branch, whether it's currently likely or unlikely, and how often the prediction failed.

### **Execution and Output**

Apart from the tool's name, the syntax hasn't changed from the regular Branch program: `branchpred app`. The tool downloads and runs the application as before, but now the output is much more informative:

```

Command Prompt
Downloading... 92%
Downloading... 100%
Press ENTER to get data, x-ENTER to quit

File D:\CodeWarrior\Examples\sce200\ui\blow\blow.c, line 181: Branch-unlikely a
ccurate only 0% of the time (12851 out of 1953393013 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\blow.c, line 245: Branch-unlikely a
ccurate only 1% of the time (1 out of 70 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\blow.c, line 253: Branch-unlikely a
ccurate only 12% of the time (1137224 out of 8982736 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 551: Branch-unlikel
y accurate only 1% of the time (1 out of 70 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 569: Branch-unlikel
y accurate only 0% of the time (0 out of 4 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 587: Branch-unlikel
y accurate only 12% of the time (1137224 out of 8982736 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 703: Branch-unlikel
y accurate only 1% of the time (1 out of 70 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 736: Branch-unlikel
y accurate only 44% of the time (65 out of 145 branches)
File D:\CodeWarrior\Examples\sce200\ui\blow\physics.c, line 802: Branch-unlikel
y accurate only 0% of the time (0 out of 9059396 branches)
x
E:\WINNT\PROFILES\Administrator\DESKTOP\stephen\branchpred>

```

*Fig. 06-04: BranchPred Output*

## Interpretation

Most likely, your application will have a number of mispredicted branches. Now what?

First, remember that you can (and should!) ignore most of them. In the sample output above, for example, there are three branches that happen millions or even billions of times, and six other branches that happen at most a few hundred times. Only those three most common ones merit any attention at all; the rest are irrelevant.

When you look at the source code in question, you'll find that these branches occur in two places: if statements and loops. Of the two, if statements are easy to optimize; loops require some tinkering.

To change the branch prediction of an if statement, just reverse the order of the clauses. For example, consider the following example:

```

if (x < y)
    a++;
else
    b++;

```

In assembly, this becomes the following pseudocode:

```

test x - y
if non-negative goto BClause
a = a + 1
goto Done
BClause:
b = b + 1
Done:

```



Looking at it this way, you can immediately see where the mispredicted branch is: the “if negative goto BClause” instruction. (The “goto Done” instruction is an unconditional branch, which by definition is always correctly predicted!) In this example, *x* is usually greater than or equal to *y*. Thus, by reversing the order of the two clauses, like so:

```
if (x >= y)
    b++;
else
    a++;
```

...then we eliminate the misprediction:

```
test x - y
if negative goto BClause
a = a + 1
goto Done
BClause:
    b = b + 1
Done:
```

Fixing loops is more of a challenge. You can’t simply reverse the direction of the comparison, because the compiler is, in effect, creating an `if` statement at the bottom of the loop for you. What you have to do is replace the C loop syntax with your own “hand-rolled” version, in which *you* write the `if` statement so that you can get the direction right. An even better option is to replace that `if` statement with a bit of inline assembly to use one of the Branch Likely instructions. Both those techniques are outside the scope of this course, however.

## Improvements

This is a much better tool than the original Branch, but it’s still not perfect.

For one thing, the output ought to be sorted, so that the first branches on the list are the ones most deserving of attention. Don’t cull the unimportant branches out, however—if the user is converting a function to assembly anyway, fixing branch prediction is basically free, and so it’s worth doing even when the gains are marginal.

A more substantive change would be to sample over time, rather than compiling a grand total of all procedures. Adopting the frame-by-frame collection mechanism of PS2Counter, for instance, the tool could collect ten frames of data at a time, then stop and let the target gather the information.

Sampling over time like this allows you analyze branch prediction in different circumstances. For example, clipping functions tend to have lots of branches, and they behave differently in different situations. Observing their behavior in real-time could tell you, for instance, that when performance matters most (when the most partially-clipped objects are on screen, let’s say), the branch prediction is at its worst. Since that doesn’t happen as often, the total branch prediction statistics

are pretty good, but you still want to flip the branches—most of the time you’d happily sacrifice some best-case performance to boost your worst-case performance.

## ***Looking for Inlining Opportunities***

Inlining is the process of taking the body of a function call and placing it directly into the calling routine. Doing so saves the cost of a function call, at the expense of making the program slightly larger.

The term is primarily associated with C++ because that language allows you to suggest to the compiler which functions would be suitable for inlining. However, it’s an important technique in any language, and it’s possible to do by hand in C or C++. Indeed, that’s the only reliable way to do it, since a C++ compiler is never required to honor your inlining suggestions.

So when would you inline? The rule is “whenever speed is more important than compactness.” In practice, this means that if you have a function that is frequently called from a small number of locations, it’s a good candidate for inlining—the frequent calls make the function-call overhead significant, while the small number of call sites means that inlining won’t increase the size of the program very much.

Unlike many optimizations, this isn’t an “either-or” tweak. You can inline a function in one location, and “outline” it (that is, leave it as a normal function call) in another. C++ does exactly that whenever you use a pointer to an inlined function—it creates an “outline” copy and gives you a pointer to it. This is why functions can be declared as both `virtual` and `inline` in C++.

## **Inliner**

This tool uses static and dynamic analysis to recommend candidates for inlining. Static analysis is used to determine which functions are small and simple enough to be viable candidates. Then, dynamic analysis tracks the number of calls made from each call site to these functions. As mentioned before, a simple function that gets called a large number of times from a small number of locations is an excellent candidate for inlining.

I’ve written this tool in C++ for a few reasons. Partly it’s an experiment—I’d never written an ATtaCK tool in C++ before, and wanted to see how well it worked. Partly it’s a demonstration, so that the hard-core C++ programmers taking this course don’t feel left out. But mostly it’s to take advantage of the Standard Template Library. This program has to create and maintain a number of lists, and STL makes that task much, much easier.

Alas, hastily written C++ is one of the gravest programming sins one can commit. I don’t recommend the C++ style of this program to anyone—it accomplishes its mission, which is the best I’ll say of it. Still, this program and the earlier TRCTool can give you some ideas on how *well-written* C++ can really benefit you in creating ATtaCK tools.

This is another tool that's not included as part of the ATtaCK distribution, so you'll need to download it from the course supplemental material folder. Once you've downloaded and unzipped the project (`inliner.mcp`), open it and then open the instrumentation tool's main file, `inliner.cpp`.

Pretty sparse, isn't it? I took the lazy-man's approach to C++ programming and stuck all the code in one big object, in this case called `RunContext`. Using objects like this has about as much in common with "object-oriented programming" as chicken feed does with chicken salad, but it's quick and easy. As I've said before, the key to getting the most out of ATtaCK is to learn how to put together tools quickly to tackle specific problems—Inliner took me all of two hours to write.

Okay, with the *mea culpa* is out of the way, look at `inliner.cpp` long enough to see what it's doing. It creates a `RunContext` object on the stack, then invokes the `InitSession()`, `BuildCandidateList()`, `Instrument()` and `Run()` methods of that object.

Now open up the *real* main file, `RunContext.cpp`.

## RunContext Object

### InitSession() Method

This function and the destructor that follows it demonstrate the virtues of writing C++ programs in ATtaCK. The `InitSession()` method handles all the boilerplate initialization code that goes into the start of every ATtaCK program. With some very slight modifications, it (and the rest of the `RunContext` object) could be made completely generic, giving you a base object from which to inherit for other programs.

### BuildCandidateList() Method

Our tool's first task is to use static analysis to build a candidate list. Toward that end, we define a `Candidate` object that will hold all the information we need regarding a particular candidate procedure. The `RunContext` object then creates and maintains a vector of these `Candidate` objects.

To do this, it iterates over every procedure in every image. A new `Candidate` object is constructed using the procedure handle and the image's ID. Then that object is queried to find out if it's a valid candidate—we'll see exactly what that entails later on, when we examine the `Candidate` object in detail.

If the object is a valid candidate, it's added to the list. Otherwise, it's deleted. In either case, the loop continues until all procedures have been checked. Remember that procedure handles, like most ATtaCK handles, last until the *program* has been released, not the image, so we're not running any risk of having the handles invalidated by exiting the loop.

### Instrument() Method

Now we need to create a list of every call site that targets one of the candidate procedures. The process to do that is very simple: Iterate over every call site, and

check to see whether its target address matches the address of a candidate procedure.

That's precisely what the `Instrument()` method does. The target address of each call site is passed in to the `FindTargetInList()` method, which scans the candidate list to find a match. Assuming the target is in the candidate list, its associated `Candidate` object is returned. This, along with the `ATtaCK` instruction object representing the call site, is used to create a new `CallSite` object, and then the call site is instrumented with a call beforehand to the `LogCall()` analysis routine.

After the iteration loop is finished and we have a count of the total number of call sites instrumented, a buffer of `atck_uint32_t` counters is allocated. This buffer is passed to a standard `Initialize()` analysis routine. The instrumented program is then written out.

We haven't finished with the `RunContext` object—there's still the download, execution and reporting code left to cover. However, to understand the rest of the tool, we need to look at the `Candidate` and `CallSite` objects and the analysis routines. Let's do that next, then come back to `RunContext`.

## ***Inliner: Instrumentation and Analysis***

### **Candidate Object**

Both the `Candidate` and `CallSite` objects are defined in `Candidate.h` and implemented in `Candidate.cpp`.

#### **Constructor**

The constructor does a substantial amount of work. Not only does it retrieve and store the procedure's important information, such as address and source code location, it also performs the check to see whether the procedure is a valid candidate.

First, we calculate how many cache lines this procedure spans. I'll explain what this means next page, and we'll discuss the cache in detail in Lesson 07.

Basically, though, cache lines are fixed 64-byte blocks of memory. To figure out how many of these blocks a procedure spans, we AND its address with 63, to see how far into the first block the procedure begins. Then we add that to the total number of instructions, add 63 (so that we still count a final, incomplete cache line) and divide by 64.

If the number of cache lines exceeds an arbitrary threshold—ten, in this case—then we assume that the impact on the cache of inlining the procedure outweighs the benefits of removing the function-call overhead.

Assuming the procedure is small enough overall, we then iterate over every instruction in it, much as we did in `PS2Counter`. If an instruction is a backward branch, then the procedure most likely has a loop, which disqualifies it—even just

a few passes through a loop consume far more time than the function-call overhead, so there's no point in inlining procedures with loops.

### **DoesAddressMatch() Method**

This inline method tests the object's address and image ID members against the specified ones. This is used to search through the list of candidates to find a match for a particular call-site's target. This is also used by an `operator==( )` method to see whether two `Candidate` objects refer to the same procedure.

### **Display() Method**

This method demonstrates another nice feature of C++ for ATtACK tools. The `Candidate` object is responsible for displaying its own profile information, so that the calling program doesn't need to know how the object is implemented or what data it contains. When we look at the `RunContext` object's display code, you'll see how clean this makes things.

By the way, if we were using stream I/O in this program, better style would have been to make this method an overloaded `operator<<( )` function instead.

## **CallSite Object**

Just as the `Candidate` object stores information about a candidate procedure, a `CallSite` object encapsulates information about a call site targeting one of those candidate procedures. Its methods can be summarized very quickly:

- The constructor fetches and stores the source filename and line number for the instruction representing the call site.
- The analysis code will gather a set of counters, one for every call site. Those counters will then be copied into their corresponding `CallSite` objects using the `SetCallCount( )` method.
- The `DoesCandidateMatch( )` method is used by the tool's display code to filter through the call-site list, so that it can print just the call-sites that target a specific candidate procedure.
- As with the `Candidate` object, the `CallSite` object is responsible for printing out its own profile information using the `Display( )` method.

## **Analysis Code**

The analysis code, found in `anal_code_PS2.c`, barely requires any explanation, since it's in C—you've seen lots of code like this. The `Initialize( )` function sets up the buffer of `atk_uint32_t` counters, while `LogCall( )` simply increments one of those counters.

## RunContext Object: Download and Execution

The download and execution code of `RunContext` is designed to be completely generic. If you reuse this object in your own ATtACK tool, you only have to override the output function, the `HandleStop()` method.

### Run() Method

This method first calls `LockDevice()`, which initializes the device connection as required. Then it downloads and runs the program. Note that the `this` pointer is passed to `atck_idownload()`. The event handler, a static method of this object, will receive this pointer so that it can call the right object's `HandleStop()` method.

Note that we don't restart the target after stopping it here, we simply return. This tool is designed to run the application and fetch the data just once—you'll see why when we look at the output code.

### LockDevice() and UnlockDevice() Methods

This code is designed to support multiple instances of the `RunContext` object—that is, multiple programs downloaded and running at the same time. I'm not sure why I did this, since I doubt very much anyone will ever actually download two programs at once. However, ATtACK supports it, so I figured I might as well. Certainly the code was easy.

The ATtACK handle to the device connection is stored in a static member. If this member hasn't been initialized, `LockDevice()` initializes the connection. It then increments a lock count, so that we know when it's safe to disconnect.

The disconnection is handled by `UnlockDevice()`, which is called by the object's destructor. If the device is initialized, then the lock count is decremented. When the lock count goes to zero, the device is disconnected and its handle set to `NULL`. This exact same mechanism, by the way, is used on the object's session and config handles as well.

Since multiple `RunContext` instances all share a single device connection, they all share a single event handler as well. Thus, the event handler is a static method. As mentioned earlier, this method receives a pointer to the `RunContext` object associated with the program that generated the event. If the event is `ATCK_TEVT_STOPPED`, this pointer is used to invoke the object's `HandleStop()` method. This mechanism could have been extended to the other event types, but they almost never have special handling associated with them.

`HandleStop()` is responsible for retrieving and displaying the profile data. By an amazing coincidence, that's all that's left to cover in this program, so let's get right to it.

## ***Inline: Output, Interpretation and Improvements***

### **RunContext Object: Output**

You’ve already seen the code that actually prints out the profile information—that’s in the `Candidate` and `CallSite` objects’ `Display()` methods. All the `RunContext` object has to do is decide which objects to print out and the order to print them in. That happens in `HandleStop()`.

### **HandleStop() Method**

First, if we don’t already know the address of the target’s buffer, we need to fetch it: Read the symbol table to find the address of the pointer, then read that address to find the address of the buffer.

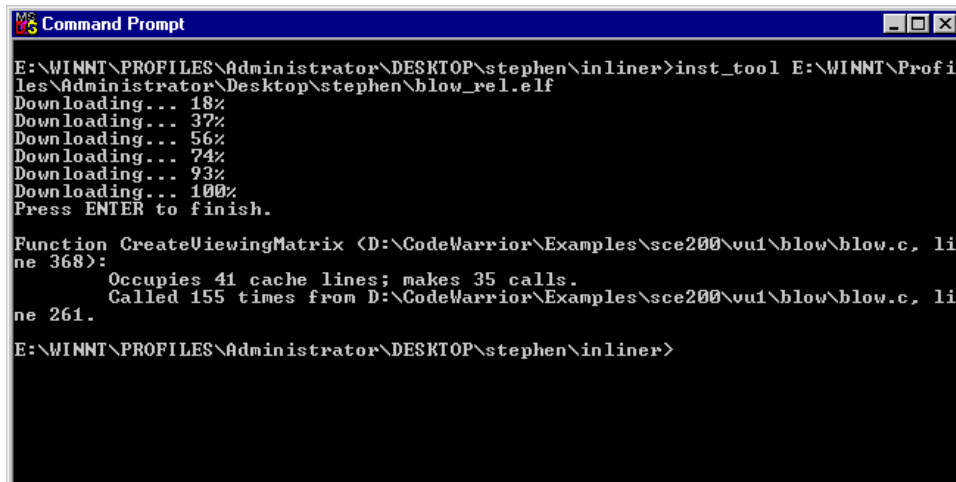
Next, upload the target buffer to our local copy. Since our call-site list consists of C++ objects, it would have been awkward at best to copy the target buffer directly to this list. Instead, we loop through the list, saving each element of the profile buffer to its corresponding `CallSite` object.

Now we sort the `CallSite` list by target address, so that we can conveniently print each candidate’s associated call sites. To do this we use the STL `sort()` function. Much like the `qsort()` function we’ve already used, this requires some form of comparison function. This being C++, a simple function pointer would be far too prosaic; instead, `sort()` takes a reference to a “functor,” an object that behaves like a function pointer by implementing an `operator()()` method. The functor in question, `CompCallSites`, is found in `Candidate.h`, and simply compares the two sites based on target address.

Once the list is sorted, the rest is easy. First we iterate through every `Candidate` object, printing its information. Then we iterate through the entire `CallSite` list, looking for the first one that matches the current candidate. Since that list is sorted, once we find one that matches, we can continue iterating, printing each one until one *stops* matching. At that point, we know that there won’t be any more sites in the list for this candidate, so we go on to the next one.

Notice that after we print out a call site, we delete it from the list. This speeds up the search for subsequent candidates, but makes it nigh-impossible to resume the program after displaying the information once. That shouldn’t be necessary. If you want to be able to collect the data in batches, though, you’ll need to replace the sorting mechanism with one that uses a separate list of pointers—*that* list gets sorted, and entries from *that* list get deleted, leaving the original call-site list untouched.

To see the result of all this code, type `inliner app` on the command line. You should get something like this:



```
Command Prompt
E:\WINNT\PROFILES\Administrator\Desktop\stephen\inliner>inst_tool E:\WINNT\Profiles\Administrator\Desktop\stephen\blow_rel.elf
Downloading... 18%
Downloading... 37%
Downloading... 56%
Downloading... 74%
Downloading... 93%
Downloading... 100%
Press ENTER to finish.

Function CreateViewingMatrix <D:\CodeWarrior\Examples\sce200\vu1\blow\blow.c, line 368>:
    Occupies 41 cache lines; makes 35 calls.
    Called 155 times from D:\CodeWarrior\Examples\sce200\vu1\blow\blow.c, line 261.

E:\WINNT\PROFILES\Administrator\Desktop\stephen\inliner>
```

**Fig. 06-05: Inliner Output**

Note that the only “candidate” here isn’t a very good one. Actually, I didn’t have a sample program with *any* good candidates, so I had to lift the cache-line threshold just to get any output at all. A real game application will have plenty of viable candidates, however.

## Interpretation

As you can see, the output becomes your roadmap for inlining: A function that is called a large number of times from a small number of locations is a prime candidate.

If you were *purely* concerned about speed, of course, you’d inline *everything*. However, as a practical matter, there comes a point where tiny speed gains are not worth making your code hundreds of times bigger. More importantly, tiny gains aren’t worth the effort of actually inlining the code—remember, your time is a lot more valuable than the CPU’s.

Actually, even if speed were your only concern, inlining would often be the wrong choice, thanks to the instruction cache. As you may recall, each cache line is a block of memory containing 16 instructions; the cache can hold 256 of these blocks. If your program is just running straight through, without any branches, then the cache is irrelevant. However, once you start looping and branching, the cache matters a great deal.

In particular, it’s extremely important that all the code from the top to the bottom of a loop, including any functions in which that loop calls, fit within the cache. That’s one of the ways inlining helps—a distant function probably won’t be in the cache, but an adjacent instruction will be. But on the other hand, if the inlined function itself blows the cache, due to its size or its own function calls, then the effort is wasted.

When you do decide to inline a function, simply adding the `inline` keyword may not help much, since the compiler is free to ignore your suggestion. Even if it decides to inline the function, it does so *everywhere*, which may do more harm



than good. We just spent a lot of effort figuring out the best places to inline, and it's a shame to have that knowledge go to waste.

To inline a function by hand, you've got two choices: C macros or assembly language. Of the two, macros are obviously much easier to write and maintain. Odds are, though, that if a function is really worth your time to inline, it's probably worth your time to move it into assembly language, where you can perform other optimizations at the same time.

## Improvements

Since inlining is so cache-sensitive, we should incorporate some measure of cache-awareness into Inliner. At a minimum, the tool should count the number of calls to different functions each candidate makes. A procedure that calls just one function a thousand times doesn't consume nearly as much cache space as a procedure that calls ten functions once each.

To do the job right, you need to have a pretty accurate model of how the cache works. The PS2Cache tool that we'll cover next lesson includes such a model. PS2Cache uses dynamic analysis to build a nearly-perfect map of the application's cache usage. Inliner doesn't need that level of accuracy, though, and could get by with static analysis. All it needs is some sense of what inlining a particular function will do to the cache usage.

There's another type of performance tweak that's related to inlining, called loop unrolling. With inlining, you make your program larger to cut down on the number of function calls you make; with loop unrolling, you make your program larger to cut down on the number of branches. A function is a good candidate for inlining if it is called a large number of times from a small number of locations; a loop is a good candidate for unrolling if it is executed a large number of times but only runs through a few passes each time.

You get the idea: Loop unrolling is really just inlining done with basic blocks rather than procedures. Turning Inliner into "Unroller" is every bit as easy as it sounds. There's even some benefit to performing both analyses at the same time, so that you get a picture of how the two interact. For instance, Inliner right now excludes procedures that have loops, under the assumption that the loop blows away the benefits of inlining. However, if analysis determined that the procedure's loop was a good candidate for unrolling, then suddenly that function might become a good candidate for inlining.

This is getting into the realm of computer-aided software engineering—there's big money in this field if you build something truly innovative! Computer processing power keeps getting cheaper, while programmer time keeps getting more expensive. The obvious solution is to use computers to write programs, and people who discover clever solutions to *that* problem are going to become rich. Someday the only code that humans write will be the code that tells computers how to write the rest of the code.

Did that make sense? Well, never mind, let's move on to our last program for this lab.

## Detecting Load Delays

The EE Core does a lot of things at once. When operating at peak efficiency, it issues two instructions every cycle. Really, though, each instruction takes several cycles to work its way through the pipeline, so the EE is often processing a dozen instructions at once. With all of that going on, one would expect to have problems trying to use a register that an earlier instruction modified.

For the most part, everything works smoothly, no matter what you do—even when two simultaneous instructions are both using the same registers, which is more than I can say for the Pentium. However, there’s still one situation that causes problems: If one instruction tries to read a register that the immediately preceding instruction loaded from memory, the EE has no choice but to wait until that load completes. This delay won’t be very long—just a few cycles—but when each instruction takes about half a cycle, a delay of even a few cycles really hurts. Combine this with a cache miss and the delay gets even longer.

The prudent PlayStation 2 programmer avoids this “load delay” whenever possible, by rearranging his code so that some other instruction comes between the load and its first use. The trouble is, it’s easy to forget to do that, since the most natural way to write code is to load the register and then use it. In fact, I’d go so far as to say that’s the *right* way to code: Most of the time, it’s more important to make the code readable and easy to write than it is to squeeze those extra few cycles out of it.

And, of course, if you’re writing C code, you don’t have much control over this. The CodeWarrior compiler does about as good a job as possible when you turn on optimization, but it just doesn’t have the intelligence that a human being does.

Thus, there will come a time when you need to scan your code for this problem. Armed with a list of every load delay, you can then evaluate each and decide how much effort it’s worth to fix. Well, lo and behold, you can write an ATtaCK tool to generate that list. You don’t even need to use dynamic analysis—you can do it all with static analysis, which means you can even scan code compiled with non-CodeWarrior compilers.

## RegStall

This one isn’t on the CD, so you’ll have to download it from the supplemental material folder. Once you do, open `RegStall.mcp`. Note that there’s only one target, since it’s just a static analysis tool—and there’s just one source file, since it’s a *simple* analysis tool!

Open `inst_tool.c`. This file, and indeed this whole program, has just two functions, `main()` and `Analyze()`. Of these, `main()` isn’t very interesting, just handling the usual initialization and command-line parsing tasks before it calls `Analyze()`. That’s where the real work happens, so let’s focus on it.

## Analyze()

The first thing this function does is allocate a buffer to hold disassembly text. The reporting code will print the disassembly of offending instructions, so that you know *which* register is causing the load delay.

Next, we allocate three register-set objects: `prevwrite`, which will store the registers loaded by the previous instruction; `curread`, to store the registers read by the current instruction; and `result`, a temporary set used to store the intersection of the two. This mechanism is the heart of the tool, so rest assured we'll examine it in detail very soon.

The outer iteration loops are boilerplate. First we iterate over the images in the application, then over the procedures in each image.

The procedure iterator in turn iterates over each procedure's basic blocks. Just before that loop, though, it clears `prevwrite`—by the time a function call transfers control to the procedure, any value loaded by the instruction in the call's delay slot will be ready for use, so the first instruction of a procedure can never suffer from load delay.

Also note that there are two pointers maintained outside the basic-block loop, `pprevinst` and `poldinstit`. The former holds a handle to the previous instruction, for the sole purpose of letting us print its disassembly out when we find a load delay. (Since writing then reading a register cause a delay, it's useful to print both the instruction that caused the delay and the immediately preceding instruction that loaded the register.)

However, keeping this old instruction handle around presents a problem. For efficiency, we're using iterator-lifespan instruction handles. Remember that the `atck_instit_new()` method takes an extra argument, a flag that says whether the instructions returned by that iterator should last for the lifespan of the image or the lifespan of the iterator. Since any production application will have many, many thousands of instructions, each of which takes up quite a few bytes within ATtACK, we really want to use iterator-life instruction handles.

We iterate over the instructions within a single basic block, and then free up the iterator. Thus, the instructions from one block would not normally be available in the next block, yet the last instruction in a basic block could easily cause a load delay. To resolve this, we don't release the old instruction iterator right away. Instead, at the end of each basic block, we release the *previous* block's iterator, and then store the *current* block's iterator in `poldinstit`.

After clearing those variables, we start the basic block iterator. As promised, for each basic block we iterate over the instructions. Skip over what we actually *do* to those instructions—I'll cover that next.

After the instruction iterator finishes, we see whether there's a `poldinstit` lying around, and if there is we release it. After that, we set `poldinstit` equal to `pinstit`.

At the end of the basic block iterator, we do that same operation again: If `poldinstit` isn't null, then we release it. Then we release the `bblock` iterator and exit to the procedure loop. When that completes we release the procedure iterator,

release the image and exit to the image loop. When that completes, we're done—we release everything and exit.

So what little work this program does is contained within a few lines inside the instruction iterator loop. Let's look there next.

## Analysis and Output

The first thing that happens within the instruction loop is that `curread` gets cleared by a call to `atck_regs_remall()`. Then `atck_inst_inregs()` adds to `curread` the set of registers read by this instruction. The set must be cleared first because `atck_inst_inregs()` *adds* the registers to the set, rather than *replacing* the set's contents with the new registers.

Next we use `atck_regs_intset()` to set `result` equal to the intersection of `curread` and `prevwrite`. The latter, remember, contains the register loaded by the previous—we'll see it get set a few lines from now. Calculating the intersection of the two sets means that if `prevwrite`'s register appears in `curread`, it will appear in `result` as well; otherwise the set will be empty.

If `result` contains one or more registers, counted by `atck_regs_num()`, then we know it's not empty, which means we know that the current instruction reads a register loaded by the previous instruction. We then print out diagnostic information: the instruction's source file, line number and disassembly, as well as the disassembly of the previous instruction. (Now you see why we had to jump through hoops to keep `pprevinst` available!) Note that it's not possible to get here without `pprevinst` being valid—this will never get called for the first instruction in a procedure—and so we don't have to check for `NULL`.

After that test, regardless of its outcome, we clear `prevwrite`. Then we check to see whether the current instruction is a load, using the convenient ATtaCK method `atck_inst_isload()`. If this instruction is a load, then we set `prevwrite` to the registers written to by this instruction—it'll just be a single register, the one loaded.

Finally, we set `pprevinst` equal to `pinst`, and continue the loop. And that's the end of the interesting code. The rest of the program wraps things up with the usual boilerplate.

## Interpretation

Here's what the output will look like:

```
MS Command Prompt
862:      lw      r3, 252(r29)
> addu   r19, r3, r6
Load delay stall in file D:\CodeWarrior\Examples\sce200\vu1\blow\physics.c, line
867:      lw      r3, 256(r29)
> addu   r20, r3, r6
Load delay stall in file D:\CodeWarrior\Examples\sce200\vu1\blow\physics.c, line
873:      lw      r3, 260(r29)
> addu   r23, r3, r5
Load delay stall in file D:\CodeWarrior\Examples\sce200\vu1\blow\physics.c, line
878:      lw      r3, 264(r29)
> addu   r21, r3, r5
Load delay stall in file D:\CodeWarrior\Examples\sce200\vu1\blow\physics.c, line
883:      lw      r3, 268(r29)
> addu   r22, r3, r5
Load delay stall in file D:\CodeWarrior\Examples\sce200\vu1\blow\physics.c, line
885:      lw      r3, 304(r29)
> addu   r3, r3, 16
E:\WINNT\PROFILES\Administrator\DESKTOP\stephen>
```

**Fig. 06-06: RegStall Output**

“Okay, it looks like this ‘load delay’ thing is happening pretty often. Now what?”

Well, there’s not much you can do in C. Make sure optimization is cranked all the way up—instruction scheduling (which includes preventing load delays) kicks in at optimization level 3. That helps, but it doesn’t make the problem go away—the sample output above was run against a program that *was* compiled with the maximum optimization.

What you’ll need to do to really solve this problem is move code into assembly. That’s obviously a drastic step, too drastic to do *just* to prevent load delays. Instead, you should be using this tool in conjunction with all the others—Inliner, PS2Cache, etc.

Look for functions that consume a significant percentage of your execution time, and that have a lot of different performance problems as revealed by the tools. Those are the best candidates for moving to assembly. The “start-up cost” of converting a C function to assembly is high, while the cost of each individual optimization—inlining, loop unrolling, instruction sequencing, branch prediction, etc.—is comparatively very low. You want to convert as few separate functions as possible, but you want each of those functions to include as many different optimizations as possible.

Also remember the 80/20 rule: There will be some small portion of your program where tweaks like this will make the most impact. Don’t waste time tweaking anywhere else. On the other hand, once you’re already tweaking a particular function into assembly, you should always use RegStall to scan it for load delays. The incremental cost of swapping instructions around to prevent load delays is very small, easily justified by the payoff.

## Improvements

I wrote RegStall in less than an hour, so there’s *plenty* of room for improvement!

First, it’s not completely accurate. When ATtaCK steps through a procedure, it does so in order of address, not in order of execution. This program assumes that

the basic blocks are executed in order, one after the other, but that's not the case—almost by definition, since basic blocks are defined by alterations in the flow of control.

When branches are actually taken, of course, then the timing of the load delay basically becomes irrelevant. So really, the only time we should consider an instruction at the start of a basic block is if the preceding basic block ends with a conditional branch. Only then—and only if that branch falls through—is it even possible to see a load delay at the start of a block.

More interestingly, we could combine this tool with others into a “megatool.” As discussed above, the best candidates for hand optimization in assembly are functions that have a lot of separate problems. You could run each tool separately and then fold their results together by hand, but if you find yourself doing that very often, you'll save time in the long run by folding the *tools* together.

The reporting for such a megatool would sort everything by function. Each problem incident would have a “score” associated with it—ideally the score would be the rough number of cycles the problem wastes. You'd then sort functions by their score, from highest to lowest, and for each function list every individual problem and the number of times it occurs.

Note that in order to get the score correct for the statically analyzed problems, you'd need to multiply by the number of times the problem was encountered. For example, let's say a load delay gets a score of 3. You'd want to insert instrumentation at the start of that load's basic block to count the number of times it happens, and then multiply the score by that amount.

In fact, this is a good improvement for SimpProf, too. If you remember SimpProf, it slows down the program somewhat, because it adds a lot of instrumentation. However, you could use these other tools' static analysis to screen out functions whose internal profile won't be very interesting: If function `x()` doesn't have any mispredicted branches, load delays or other internal problems, then you're not going to be optimizing it, and if you're not going to optimize it, you probably don't need to profile it at all.

Ultimately, though, it all comes down to your personal optimization technique. Keep in mind one of my guidelines from Lesson 05: Don't ask questions when the answer doesn't matter. If you're not looking for functions to hand-optimize with assembly, then don't bother gathering detailed lists of potential assembly tweaks!

## Just One More!

That concludes our penultimate lesson. Next up is Lesson 07, another lab, in which we'll look at memory problems and memory analysis tools.

## Lesson 07: Analyzing Memory and Cache Usage

---

Our final lesson is another lab, covering four tools to solve common memory problems. As usual, we'll talk about improving these tools as well as how to use them as-is.

### ***Catching Misaligned Memory Accesses***

In general, memory accesses on the PlayStation 2 must be aligned on a multiple of their size: word loads must be aligned on word boundaries, halfword stores on halfword boundaries, and so forth. Most chips have this restriction. Accesses that violate this rule generate an exception—a very powerful detection tool!

However, the EE's "multimedia extensions" to the MIPS instruction set work a little differently. Those extensions feature two instructions, `LQ` (Load Quadword) and `sq` (Store Quadword) that operate on 128 bits at a time, a very handy feature for game operations

Like the EE's other memory-access instructions, these only work on multiples of their size, namely 16 bytes. However, *unlike* the other instructions, invalid addresses don't generate exceptions. Instead, the bottommost four bits of the address simply get masked off.

At first glance, this may seem like a helpful feature: "Neat, if I screw up, the application won't crash!" But in fact it's not very helpful at all. After all, if the bits being masked off are non-zero, you've probably made a mistake, either in your algorithm or in your memory allocation. Now, when would you rather find out about that mistake: the minute it happens, or after a week of desperate debugging and analysis as you try to figure out why the first three pixels of your textures are always random colors?

### **PS2Quad**

That's where our first tool of this lesson, PS2Quad, comes in: It looks for and catches these misaligned accesses, generating a "pseudo-exception" by halting the system.

The concept behind PS2Quad is extremely simple. The instrumentation loop looks for every `LQ` and `sq` instruction, and inserts a call to analysis code before each one. The analysis routine check's the effected address' bottom four bits, and halts the target if any of them are non-zero.

PS2Quad is located in the Examples folder of your ATtACK installation—the full path is probably `C:\Program Files\Metrowerks\ATtACK for PS2\Examples\PS2Quad`. Within that folder is the `Inst` folder, containing the instrumentation-tool project `ps2quad_inst.mcp`. Open that project, then open its only source file, `ps2quad_inst.c`.

## Instrumentation

The `Instrument()` function uses a basic iteration loop to step through every image in the program, then every procedure in each image, then every basic block in each procedure, then every instruction in each basic block.

For each instruction, we retrieve its “pseudo-opcode.” This is an ATtACK-defined enumerated type that indicates what the instruction basically is. By comparing this value to `ATCKPS2_OP_LQ` and `ATCKPS2_OP_SQ`, we check to see whether the instruction in question is one of the quadword access instructions, `LQ` or `SQ`. While we’re at it, we make sure the instruction can be instrumented with `ATCK_EFFADDR`—if it can’t, no point in proceeding further with it.

Assuming this is an instrumentable quadword instruction, we insert an instrumentation call before it. This instrumentation calls the `QuadCheck()` analysis routine passing it the image ID number, the address and the effected address of the instruction. We’ll see how those are used in a moment.

That’s all the instrumentation loop has to do. After that, the loop finishes, the instrumented application is written out, and the tool downloads and runs the program. Let’s see what the analysis code will do—open up `Anal\ps2quad_anal.mcp`, then open its only source file, `ps2quad_anal.c`.

## Analysis

There’s just a single results structure, `QuadDat`. The tool is going to halt the program immediately upon a misaligned access, so there’s no need for a buffer of multiple results.

The `QuadCheck()` routine receives the image ID, the instruction’s address and the address that instruction is trying to access. If that address ANDed against `0x0F` is zero—that is, if all of the bottom four bits are set—then the address is quadword aligned, and the routine simply returns.

However, if it’s not, then we need to raise our “pseudo-exception.” The values passed into the routine are stored in the `QuadDat` structure, the target is stopped and the program outputs the diagnostic information.

## Output

The `RunIt()` and `HandleEvt()` functions are more or less cut-and-paste from other ATtACK tools, so we can ignore them. To the extent that anything interesting happens in this tool, it happens in `PrintMisalign()`.

First, this function declares the `quaddat_t` structure to ATtACK, so that it can read the entire structure at once. It then gets the address of the target’s structure, and uploads its contents into the local copy.

Next, the function needs to find the offending instruction. It can look up the instruction by address, but first it needs an image handle. We couldn’t pass an image *handle* to the analysis routine, but we did pass an image ID number, which is now in our local copy of `quaddat`.



The `GetImg()` function returns an image handle based on an image ID number. It does this simply by iterating through the images in the program until it reaches the specified ID.

Once we have the instruction handle, we can start dumping out information about the misaligned access: the instruction type (LQ or SQ), the source filename and line number and the offending address.

To make the output more useful, we then disassemble a forty-line window around the instruction. If you'll remember back to Lesson 02, we discussed iterating from one object to its siblings. The best way to do that is to step out to the parent, then iterate across the parent. We do that here by getting the basic block that contains the instruction, then the procedure that contains the basic block. This procedure is then passed to `PrintDisasm()`, a helper function.

`PrintDisasm()` disassembles a 40-byte “window,” five instructions on either side of the offending quadword access. This window begins at `addr_start` (the target instruction's address minus 20) and runs to `addr_end` (`addr_start` plus 40).

First, the function prints out the procedure's name, and allocates a buffer to hold the disassembly text. Then it iterates through every basic block in the procedure, then through every instruction in each basic block. If an instruction's address falls within the window, its disassembly text is fetched and printed.

And that's all there is to PS2Quad. Normally I'd show you the tool's output, but like they say, no news is good news—most programs won't have this particular bug, so for most programs there *is* no output. Certainly I couldn't find a sample program that had the problem.

## Improvements

This tool's almost too simple to improve! To be really useful, though, it needs to run all the time—the nature of the quadword access problem means that you won't usually know you need to look for it. The overhead of PS2Quad is very small, so you should probably fold this into the other tools. That way, whenever you're performing one form of analysis, you're also watching out for this bug,

## ***Hunting Down Memory Leaks***

What's the worst bug you've ever had to track down? More than half the time when I ask people that, they tell me about a memory leak. Leaks are notorious for being easy to commit and hard to track down.

To have a hope of finding the leak in hours rather than days (or weeks!), you need a good memory-leak detection tool. The programmer simply not releasing the memory causes some leaks, and inspecting the code can spot those. More often, though, the leak occurs because a pointer gets modified (and thus can no longer be used to release the block), or worse yet gets overwritten entirely, orphaning the block.

Following every memory pointer around for its entire lifespan, watching for it to change, is simply not something human beings can do. Computers are great at it, though, and tools to do exactly that have been available on the PC for many years now. These tools are indispensable, but they generally don't exist on game platforms.

So let's write one! ATtaCK gives us all the pieces we need to track memory allocations. The process is simple:

- Log every call to `malloc()`—the bytes requested and the pointer returned
- Log every call to `free()`
- Run through the logs, eliminating matched pairs of `malloc()/free()` calls
- Anything left in the list is a leak

The thorniest problem is managing these logs. At eight bytes per call, they can get big very quickly. Many applications use `malloc()` to allocate large numbers of small structures, so in a worst-case scenario the log could consume as much memory as the application's data!

To keep that from happening, all the analysis and log management should be done on the host. The target should just gather the data, as quickly and minimally as possible. When the target's buffer is full, it should halt itself and let the host read and clear the buffer. That way, most of the logs are stored in the host's RAM rather than the target's.

This tool—we'll call it Plumber, since it stops leaks—isn't included in the ATtaCK distribution, so you'll have to get it from the supplemental material folder. Once you've unzipped the project, open up `plumber.mcp`, then open `insttool.c` from the `Instrumentation Tool` folder.

## Navigation and Instrumentation

As usual, we'll skip the typical initialization code and go straight to `Instrument()`. This function performs the usual steps of declaring our data structure and analysis routines, then iterating through every call site in every procedure in every image. For each call site, the name of the target function is retrieved.

If that name is "malloc", then obviously this is a call to `malloc()`, so we instrument it. Conceptually, we just want to log the bytes requested and the pointer returned. However, in practice, we can't count on registers not changing across the function call. Therefore we need to read the bytes requested *before* the call, and the pointer returned *after* the call.

To do this, we add instrumentation before the call that passes the contents of register `ATCKMIPS_REG_GPR4`. In assembly code, this register is known as `a0`, and it typically holds the first argument of C function calls. In this case, that argument will be the number of bytes to allocate. Next, we add instrumentation after the call site to get the pointer that `malloc()` returns. This pointer comes back in register `v0`, which in ATtaCK-speak is `ATCKMIPS_REG_GPR2`.

For both instrumentation calls, we also pass the callsite ID. Notice that this ID starts at 1, not 0. Later on, we'll use a callsite ID value of 0 to indicate a list entry that's already been resolved.

If the call target's name is "free", then this is a call to `free()` and our job is much easier. `free()` doesn't have a return value, so all we need is instrumentation before the call to log the value of `ATCKMIPS_REG_GPR4`. As with `malloc()`, we also pass in the callsite ID.

## Analysis Routines

Plumber's analysis code is complicated because of that split between "before" and "after." If we were interrupted in between these two instrumentation calls by another allocation, there's no telling what state the list would wind up in. Certainly the profile would become useless. Thus, we need to ensure that no other analysis routines can possibly run in between the two calls that surround `malloc()`. Sure enough, ATtACK gives us a mechanism to handle this—mutexes.

### Initialize()

As usual, this gets called at the start of the application. And as usual, it receives a pointer to a buffer allocated by ATtACK, saving that pointer in a known variable so that both the host and target can find it.

Here, though, `Initialize()` also receives the number of elements in the buffer. We need to track the buffer size in the analysis code, because when we've filled the limited buffer space we'll stop and let the host read and clear the buffer. That way, we don't have to allocate a buffer large enough to hold every memory allocation.

Finally, we initialize the mutex variable. All this requires is passing the variable's address to `atcktarg_initlock()`. This initialization must be done in code that's guaranteed to be executed single-threaded. Since analysis routines called using `atck_callbefore()` run before the program even starts, this fits the bill.

### BeforeAlloc() and AfterAlloc()

These are really just one routine split into two halves.

Before the allocation call, we lock the mutex. The subsequent `while` loop looks strange, but ignore it for a minute—suffice it to say that it's ensuring the buffer has room. In any case, by the time we make it past the loop, we know that we're holding a lock to the mutex. That means that if the application is allocating memory in another thread (and thus locked the mutex in that thread before us), that thread will have to finish first before we can proceed.

Once we're past that, we set the bytes member of the current buffer entry equal to `caller_a0`, which are the contents of register a0 passed to us by ATtACK. This is the number of bytes requested. We also save `callsiteID` in the structure's `callsiteID` member. Then we return, which allows the system to proceed.

The application branches to `malloc()`, allocates the block, and returns, at which point `AfterAlloc()` activates. The pointer to the allocated block comes back in `v0`, passed to us by `ATtACK`. This value is stored in the `ptr` member of the current structure. Then, the log entry now being complete, we increment the buffer index. That marks the end of our critical section, so we release the mutex.

At this point, if the buffer is now full, we halt and let the host read and clear the buffer. Note, however, that if another thread is waiting on the mutex, that thread will execute before we get to this call—it wakes up the moment `atcktarg_unlock()` is called. That's why `BeforeAlloc()` had to make sure the buffer wasn't already full—in effect, the call to `atcktarg_lock()` could result in the buffer index incrementing.

Look back at `BeforeAlloc()`. Now we can understand what's going on in that `while` loop. If the buffer is full, then we need to halt and allow the host to clear it. This requires releasing our mutex, since we can't halt while we've got a mutex locked. Then the loop halts the target. Presumably the host clears the data and resumes the target, at which point we lock the mutex again. Thus, it's not possible to leave this loop without a) the buffer having room, and b) holding the mutex.

## **BeforeRelease()**

Calls to `free()` are easy, since `free()` has no return value. Thus, this function can be done all at once rather than split. We still need to grab the mutex in case of multithreading, though, since our analysis routine can still get interrupted. You can see that this function is literally just `BeforeAlloc()` and `AfterAlloc()` pasted together. The one difference is that `bytes` member is set to zero, indicating that this call is a release.

That's the entirety of the analysis code. In spite of having to jump through one extra hoop, it's still pretty simple. Next up: download, execution and output, which is where all the heavy lifting gets done.

## ***Plumber: Execution and Output***

### **Execution**

The `Run()` function is pretty much the same routine you've seen before. It downloads and runs the application immediately. The user can pause the target and fetch the buffer by hitting `ENTER`, or he can simply let it run—when the buffer is full, as we've seen, the target will halt itself. Since the analysis routines are protected by a mutex, the host won't be able to halt the target in the middle of a memory call, so the buffer is guaranteed always to be valid.

In either case, whether halted by the user or by the analysis code, when the target stops it generates an event. The event handler then calls `FetchAndClear()` to fetch and clear the contents of the buffer.

## FetchAndClear()

First, this function reads the contents of the analysis code's global variables `TheBufferIndex` and `TheBufferPtr`. The first gives the number of log entries currently in the buffer, while the second is the address of the buffer itself. Then the function reads that number of entries from the specified address. These new log entries are stored in a temporary buffer, the one originally allocated and passed to `ATtaCK` in `Instrument()`.

The temporary buffer needs to get appended to our permanent results buffer. First, though, that buffer has to be expanded to make room for the new entries. We calculate the number of bytes the buffer already contains, and the number of bytes the buffer needs to grow by. If the buffer hasn't been allocated yet, then we create it using `malloc()`; otherwise, we expand it using `realloc()`. Now we copy the new bytes into the buffer, starting at the end of the existing bytes, and add the number of new entries to the count of existing entries.

Finally, we clear the target's `TheBufferIndex` variable. That's all we have to do to "clear the buffer." We don't need to zero-out the buffer itself: Unlike most of the other programs, this tool isn't counting, and so it's not important that the counters be reset to zero.

By the way, describing this process has made me realize that I'm wasting time. I don't need the temporary buffer at all, because I can just read directly into the permanent buffer, offset by the number of items already present. However, there's no point in changing it now, since the code works. The cost of the extra copy operation is almost totally irrelevant.

This presents a good lesson: `ATtaCK` tools should emphasize speed of programming over efficiency—at least on the host side—because you'll be running them on developer machines with plenty of processor speed and RAM. Hardware is cheaper than programmers; so trading hardware time for programmer time is always worth it.

As its last act, `FetchAndClear()` resumes the target. Eventually the user will hit `x-ENTER` to kill the application, at which point control returns from `Run()` to `main()`, which then calls `AnalyzeResults()`, the reporting function.

## Output

The output code is much more involved than for other tools, because it actually performs a portion of the analysis. The idea here is to sort the log by allocated address, then in order of occurrence. In the resulting list, any memory allocation that is properly released will appear as a `malloc()` immediately followed by a `free()`. If a `malloc()` appears by itself, then we know that there's no `free()` associated with it and it's a leak.

To perform the sort, we first create an array of pointers to elements in our results array. This level of indirection allows us to sort the array without disrupting its internal order. Since the array is already sorted by the order in which the calls happened, we don't want to lose that order—we need to know whether the

`free()` for a particular address happened before or after that address was allocated!

We use the standard C library function `qsort()`, which is reasonably efficient but more importantly is really easy to use. This function sorts an array in place, taking a pointer to the buffer, the number of entries to sort, the size of each individual entry and a pointer to a comparison function.

The comparison function we use here is

`CompareMemLogsByPtrThenSequenceID()`. Describing the function almost takes less time than *naming* it: The entries are sorted by ascending value of `ptr`; if two entries have the same value, then they are sorted by their position in the original list (so that a `free()` always comes after its associated `malloc()`). The mutexes in the analysis code guarantee that entries will always be added to the log in order, so the order of the original list is the order the calls happened chronologically. That's why indexes into the original list are referred to as "sequence ID."

With the list sorted, we can now run through it to find mismatched allocations and releases. First, every allocation gets added to a running tally of allocated bytes and allocated blocks. Then, for any `malloc()` immediately followed by a `free()`, we add the allocated bytes to the "bytes freed" counter and increment the "number of freed blocks" counter. Then we clear out both the `malloc()` and the `free()` from the sort array by setting their `callsiteID` members to zero.

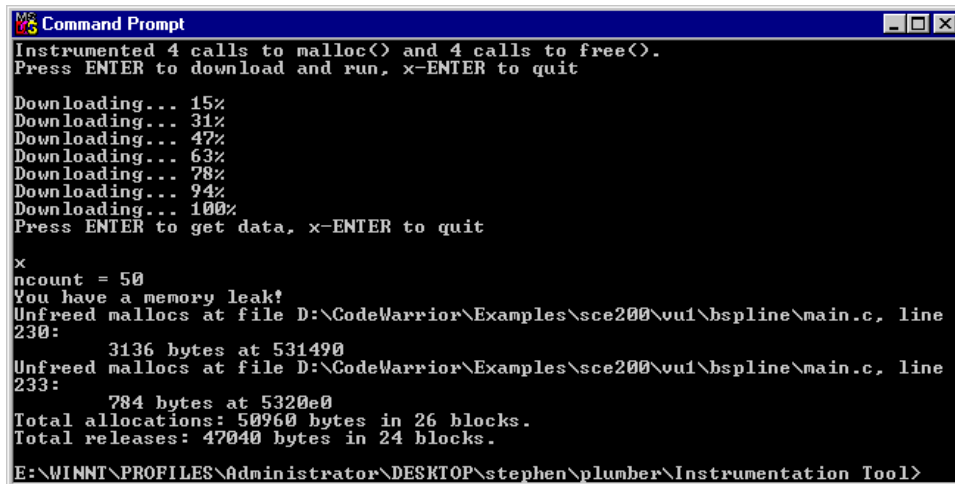
Once all that is done, we see whether our "blocks allocated" counter matches our "blocks freed" counter. If so, then we congratulate the user and quit. Otherwise, we report each offending allocation.

To do this, we sort the list again, this time by `callsiteID`. The comparison function for this is called (take a deep breath!)

`CompareMemLogsByCallsiteIDThenSequenceID()`. It sorts the list in order of ascending `callsiteID`. All entries for a given `callsite` are then sorted by their position in the original list.

Now we reiterate over the program to get every `malloc()` and `free()` `callsite`. For each site, we then increment through the results list until we find a `callsiteID` greater than or equal to this site. If the first ID we find is greater than the current one, then we can move on to the next `callsite`—thanks to the sort, we know that there aren't any entries in the entire list for this site.

But if we find an ID that equals the current one, then we know that this site was responsible for one or more memory leaks. (Remember, the matched `malloc()/free()` pairs had their `callsiteID` values zeroed out.) We then print out the diagnostic information for the site, listing every entry for this site still in the list. Here's what the output looks like:



```
Command Prompt
Instrumented 4 calls to malloc() and 4 calls to free().
Press ENTER to download and run, x-ENTER to quit

Downloading... 15%
Downloading... 31%
Downloading... 47%
Downloading... 63%
Downloading... 78%
Downloading... 94%
Downloading... 100%
Press ENTER to get data, x-ENTER to quit

x
ncount = 50
You have a memory leak!
Unfreed mallocs at file D:\CodeWarrior\Examples\sce200\vu1\bspline\main.c, line
230:
    3136 bytes at 531490
Unfreed mallocs at file D:\CodeWarrior\Examples\sce200\vu1\bspline\main.c, line
233:
    784 bytes at 5320e0
Total allocations: 50960 bytes in 26 blocks.
Total releases: 47040 bytes in 24 blocks.
E:\WINNT\PROFILES\Administrator\DESKTOP\stephen\plumber\Instrumentation Tool>
```

*Fig. 07-01: Plumber Output*

Note that most programs will have a “memory leak.” If you’re going to keep a block of memory until the program terminates, there’s no pressing need to release the memory, so many people don’t. However, trying to free every block you allocate has the advantage of eliminating that chaff during debugging. In any case, armed with a list of exactly which allocations never get freed, you shouldn’t have any problem sifting out those “intentional” memory leaks from the unintentional ones.

There’s another type of error this tool reports, unallocated releases. This occurs when `free()` is called with an invalid pointer. The most likely cause of this is incrementing your only copy of an allocated memory pointer.

## Monitoring Stack Depth

On the PC, memory is relatively easy to come by. More to the point, the consequences of running out of memory aren’t too drastic—some unused part of the operating system gets paged out to disk. At worst, your app’s performance dies, but at least the app itself doesn’t crash.

On the PlayStation 2, memory is much more of a premium. First, you’re trying to keep more data in memory, because reading from the DVD is slower than reading from a hard drive. Second, when you run out of memory, that’s it, game over—there’s no virtual memory manager to cover your tracks. So every byte is precious.

On the PC, you generally don’t worry about the stack. If, during debugging, you ever run out of stack space—assuming it’s not caused by an infinite-recursion bug—you simply increase the stack allocation and go on with your life.

On the PlayStation 2, it’s not so simple. You want the stack to be as small as possible, but you also can’t afford to ever run out of stack space. The constraints are especially severe if you want to put your stack in scratchpad RAM, of which you only have 16K!

This analysis tool instruments your application to determine the deepest stack it ever uses. During development, you can allocate an overly large stack while you collect this depth information. Just before release, you trim your stack to the minimum you need (plus some safety margin)

## DeepStack

This project is found in the Thrill Seeker Tools folder (probably `C:\Program Files\Metrowerks\ATtACK for PS2\Thrill Seeker Tools`), in the subfolder `StackDepth`. Open `inst\stackinst_ps2.mcp`, then open `stack.h`.

This file defines the data structure that our analysis routines will gather. It has two members, a procedure ID and an address. Not only does this tool measure how deep the stack gets, it also tells you exactly which sequence of calls resulted in the deepest stack.

Now open `stack_inst.c`. The `main()` is a boilerplate, handling initialization, allocation of two buffers—one for deepest stack, one for current stack—and driving the rest of the program. The work gets done in the functions `DoInstrument()`, `DoRun()` and `DoPrint()`. Before we look at them, though, we should probably look at the analysis routines.

## Analysis

Open `anal\StackAnal_ps2.mcp`, then open `stack_anal_ps2.c`.

As mentioned, there are two buffers, one for the current call stack and one for the deepest call stack. In addition, there are counters to store the number of calls in each stack, and variables to hold the deepest stack address.

The `Initialize()` function is pretty standard. It receives buffer pointers from ATtACK, stores them and zero out the counters. Note that stacks start at high addresses and build *downward*, so the starting values for the maximum-depth variables are the highest possible values.

The `RecordDeepestStack()` function is a utility routine that copies all the “current” variables to their “deepest” counterparts.

The `ProcedureCall()` function gets called before every procedure call, receiving the procedure ID and the function’s stack pointer *after* the frame is set up. The first thing that happens in this function is the address is checked to see whether it’s in the region reserved for interrupt handlers. An interrupt handler might call back into the target application, at which point the target function would use the interrupt handler’s stack rather than the application’s original stack. We’re not interested in measuring the interrupt handler stack, and since its address will be much lower than the application stack’s address, we need to filter it out.

If we pass that test, though, we add the current stack information to our call stack. The current address is set to the new stack address. The procedure ID and stack address are pushed into our buffer, and the stack depth is incremented. Finally, if the new stack address is less than the deepest one to date, we call `RecordDeepestStack()` to save this stack as the deepest one.



The `ProcedureEnd()` function has only one task to perform: It decrements the stack depth as the stack frame gets released at the end of the procedure.

Finally, the `StackUpdate()` function gets called whenever any instruction modifies the stack pointer (register 29). The address is recorded, but the stack *depth* isn't increased, since we're still within the same procedure. Again, if the new address is below the heretofore deepest address, we call

```
RecordDeepestStack().
```

Armed with this understanding of the analysis routines, you could probably figure out the instrumentation tool on your own, but we'll go ahead and look at it.

## Instrumentation

`DoInstrument()` first declares the `Initialize()` routine, adding it to the start of the application. It also declares the other routines so that they're available while we iterate over the application. Then it allocates a register-set object, which we'll need to figure out whether any given instruction modifies `ATCKMIPS_REG_GPR29`, the stack register.

Next we iterate over every image, then over every procedure within each image. Each instrumentable procedure gets a call beforehand to `ProcedureCall()`, which receives the procedure ID and the contents of register 29. Every procedure also gets a call afterwards to `ProcedureEnd()`, which doesn't take any arguments.

Now we iterate over the procedure's basic blocks and then over each block's instructions. For each instruction, we clear the register set and use `atck_inst_outregs()` to fill it with every register this instruction modifies. If register 29 is a member of the resulting set, then this instruction modifies the stack pointer, and so we instrument it with a call *after* to `StackUpdate()`, passing the new value of register 29 resulting from the instruction. (A call *before* would pass the original value of register 29, not the new one.)

The iterator loops finish normally, with each image getting written out since this is a dynamic analysis tool. Finally the entire program gets written out, and `DoInstrument()` returns to `main()`, which will then call `DoRun()`.

## Download and Execution

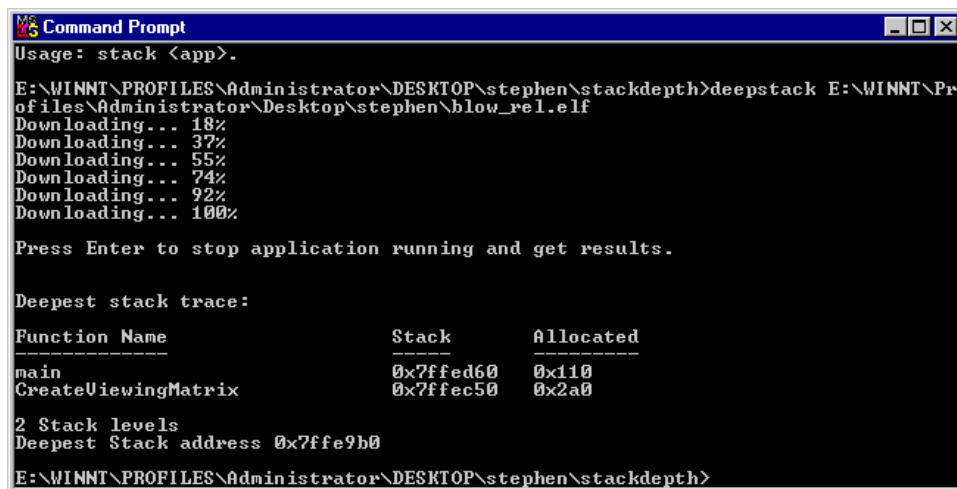
`DoRun()` is just more boilerplate that you've seen before. It has one interesting feature, though—it was originally written for earlier versions of the ATaCK API, and so in the comments you can see some unfamiliar function calls. That's just of historical interest. In terms of present-day functionality, `DoRun()` just launches the application immediately and loops until it's finished.

## Output

Finally, `DoPrint()` is called to display the results. This function does something a little more elaborate than most output routines.

First, it iterates across the application to put all the function names into an array. This array can then be used to match labels to procedure IDs, making the output more human-readable.

For each level of the application's deepest stack, DoPrint() displays the function name and the stack address. It then subtracts this level's stack address from the previous level to display how many bytes were allocated.



```
Usage: stack <app>.  
E:\WINNT\PROFILES\Administrator\DESKTOP\stephen\stackdepth>deepstack E:\WINNT\Pr  
ofiles\Administrator\Desktop\stephen\blow_rel.elf  
Downloading... 18%  
Downloading... 37%  
Downloading... 55%  
Downloading... 74%  
Downloading... 92%  
Downloading... 100%  
  
Press Enter to stop application running and get results.  
  
Deepest stack trace:  


| Function Name       | Stack     | Allocated |
|---------------------|-----------|-----------|
| main                | 0x7ffed60 | 0x110     |
| CreateViewingMatrix | 0x7ffec50 | 0x2a0     |

  
2 Stack levels  
Deepest Stack address 0x7ffe9b0  
E:\WINNT\PROFILES\Administrator\DESKTOP\stephen\stackdepth>
```

Fig. 07-02: DeepStack Output

## Interpretation

Okay, so what should you actually *do* with the information that your application consumes 944 bytes of stack space? If you're not pushing the limits of system RAM, then do nothing and leave well enough alone.

Most likely, though, you need all the space you can get. Or, if your stack is small enough, you want to move it to scratchpad RAM, to speed up the application and save cache space for more important data. The linker control file allows you to specify how big (`_stack_size`) and where (`_stack_addr`) the stack should be. However, that's an advanced subject outside the scope of this course—you'll have to read the LCF documentation that came with CodeWarrior for more information.

## Improvement

This is another one of those programs that are pretty hard to improve: It does what it does. The best possible "improvement" would be to combine this program with other ones, so that no matter what tool you're running, you've also got the stack information.

You really need to *always* monitor stack depth, so that when it comes time to trim the stack to the minimum size you need, you're confident that you've covered every possible situation. Toward that end, even if you don't merge this tool with others (such as SimpProf), you could at least add code-coverage to this tool, so that at the end of the run it tells you which routines weren't called. Your test

session must run every routine in the app at least once for you to have any confidence at all in your stack depth figure.

## **Analyzing Cache Usage**

Our last sample program is probably the single most valuable one of the bunch. The other programs are all useful, but you could write them yourself without too much difficulty.

This program is much more advanced. It uses a mixture of static and dynamic analysis to actually simulate the behavior of the PlayStation 2's data and code caches, so that it can tell you exactly where and when your application is missing the cache.

The importance of good cache usage cannot be overstated. Poor cache usage can kill the performance of even the most well-optimized assembly code; savvy cache usage can make even unoptimized C code run fast enough for release.

The cache isn't the most important aspect of PlayStation 2 optimization—that honor belongs to keeping the VU0 and VU1 pipelines full, a topic outside the scope of this course. But it's the most important aspect of EE Core optimization, and certainly if your EE core code is slow, you're going to find it very difficult to keep the vector units happy.

## **PS2Cache**

It's much easier to say what this program does than to explain how it does it. The analysis routines simulate the behavior of the cache. Instrumentation calls this cache simulator every time an instruction reads from or writes to memory.

To simulate the operation of the instruction cache, normally we'd need to add instrumentation every time an instruction is executed. However, thanks to ATtaCK's static analysis, we can simply add instrumentation at the head of every basic block. Within the block, we know that every instruction will be executed in order, so we can figure out ahead of time what the cache behavior will be.

Every time the simulator determines that the cache is hit or missed, it increments the appropriate counter for the procedure that caused the cache activity. Actually, it'd be a simple matter to increase the resolution, so that it logs the cache performance of every instruction.

Aside from using the frame-at-a-time collection mechanism that we saw already with PS2Counter, the instrumentation tool doesn't do anything out of the ordinary. All the heavy lifting is done in the analysis routines—understand those and you understand the program. But to understand those, you need to understand how the cache works.

# The PlayStation 2 Caches

## Instruction Cache

The PlayStation 2 has two caches, a 16K one for instructions and an 8K one for data. For now, let's just look at the instruction cache—the data cache works exactly the same, apart from being read/write instead of read-only.

The instruction cache has 128 “lines,” each of which consists of two “ways”; each way holds 64 bytes. This adds up to  $128 \times 2 \times 64$ , or 16K. You can think of a way as a small buffer of super-fast RAM. When the EE executes instructions, it first checks to see whether the instruction is in a cache way. If it is, it reads from that rather than memory. Otherwise, it loads 16 instructions into the way and continues.

So what are lines? Lines are how the EE organizes the ways. Imagine if you were writing software to simulate a cache—not coincidentally, we're about to! How would you check to see whether an address was already cached? The slow way would be to store, for each way, the address it's caching, and simply scan the list of 256 ways to see whether the requested address was there. That might be good enough for software, but it's wildly impractical in hardware.

Instead, the EE organizes the ways into an array of 128 pairs of ways. It then indexes this array by shifting the address right six bits (dividing it by 64) and ANDing the result with 127. What this means is that a given address will always be found in one of the two ways of a specific cache line, if it's to be found in the cache at all.

This also means that two addresses can both expect to occupy the same cache line. For example, the addresses `0x100EE00` and `0x16BCE00` can both occupy just one line, line 56. That's where the ways come into play: If way 0 of line 56 already holds `0x100EE00`, then `0x16BCE00` will get stored in way 1. If both ways are full, then the line least-recently filled will be reused.

## The Data Cache

As mentioned, the data cache is very similar to the instruction cache. (Almost as if they planned it that way!) The data cache has just 64 lines, still of two 64-byte ways each. Data lines are indexed by shifting right 6 bits and then ANDing the result with 63.

Just like the instruction cache, the data cache stores the mapped address, a “least-recently-used flag” bit and 64 bytes of data for each way. Since the data cache is read/write, it also stores a dirty flag bit. The dirty flag indicates that the contents of the cache way do not match the contents of the mapped RAM.

When would this happen? Basically anytime you write. When you write to memory, the cache first makes sure the destination is mapped to a full cache way, as for a read. Then it writes the data to the cache, but *not* the mapped RAM.

Instead it just marks the way as dirty. As long as that way remains mapped to the same address, it remains dirty. When that way gets mapped to a different address due to a read or write miss, however, the way is written to memory, in an

operation known as “writeback.” If your app never misses the cache again, the data will never get written to RAM. (But how would you know?)

Oh, and in case you’re wondering: The scratchpad RAM isn’t cached, because it’s just as fast as the cache. (It’s essentially *part of* the cache.) When you consider that the scratchpad is twice as big as the entire data cache, you can understand why it’s so important to use the scratchpad properly. And in addition to the scratchpad RAM, you can lock individual ways in the data cache to turn them into 64-byte blocks of scratchpad RAM. But both those topics are beyond the scope of this lesson.

## PS2Cache

So armed with our new understanding of how the cache operates, let’s see how PS2Cache simulates this behavior.

### Analysis

Open Examples\PS2Cache\Anal\ps2cache\_anal.mcp (by default located in C:\Program Files\Metrowerks\ATtACK for PS2\). Within that, open ps2cache\_anal.c.

Once we’re past the boilerplate, the first thing we see is a `way_t` structure. This is the data structure that simulates a single way. We described each way as holding the address, a bit-flag for whether it was used recently and 64 bytes of data. Well, we’re just concerned with the cache’s overall behavior—we’re not actually caching anything—so we don’t have the 64 bytes of data. But we’ve got the address, called `pfn_v`, and the LRF bit-flag, `r`. (The labels make sense if you read the hardware documentation for the cache. Don’t try to understand them right now.)

In addition, there’s the aforementioned dirty-flag bit for the data cache, as well as a bit to indicate whether a way is locked. The dirty flag is used for simulating writebacks; the locked flag is ignored, since this version of the tool doesn’t simulate cache locking. Both flags are ignored for the instruction cache, of course, but for convenience and clarity we use the same way structure for each cache.

These structures are grouped together into a two-element array, which then becomes the `line_t` structure. The caches themselves, `ICache` and `DCache`, are arrays of these structures (with 128 and 64 elements respectively).

Next we’ve got the usual stuff any analysis tool needs: a pointer to a results buffer, and the same countdown and enable variables that we saw in PS2Counter back in Lesson 05. If you don’t remember that program’s details, you should look there for more information.

The `Initialize()` function works as advertised, setting up the variables and clearing both cache arrays. Notice that `pfn_v` is set to 1. `pfn_v` actually holds the address shifted left one bit. The bottommost bit is zero if the address is valid—i.e., if the way is filled with data—or 1 if the way is empty. The actual PlayStation

2 cache uses a very similar but not identical mechanism, which we don't have to worry about—this is a perfectly fine simulation of that system.

The `Enable()` and `Disable()` routines are cut-and-paste from `PS2Counter`. The important thing to remember is that our analysis routines will check the boolean variable `Collect`; if that is false, then we don't gather any data.

This is the last I'm going to talk about `Collect`; you'll see it in every other routine, but I'm going to act as though it's always true. Let me just say this: The cache simulation runs all the time, regardless of `Collect`, as it must to be accurate. `Collect` just determines whether we're actually going to measure the results of the simulation this frame.

`DWrite()`, `DRead()` and `IRef()` simulate the behavior of data writes, data reads and instruction reads, respectively. Let's look at them in the reverse order, since that's actually how they make the most sense.

## **IRef()**

This routine receives a procedure ID (for indexing the results buffer), the address of the first referenced instruction shifted right six bits, and the number of instructions that will be referenced in this way. Normally that number is 16, but if a basic block ends before the end of the 16-instruction way, the number will be less.

As promised, the `ICache` array is indexed by shifting the referenced address right six bits and ANDing it against 127 (0x7F). The instrumentation tool handles the shift-right for us, so all we need to do is the AND.

The “tag” is the address, stored in the format used by the cache itself. Each cache way stores not the full mapped address, but just that address's page frame number. (Page frame numbers are abbreviated PFN. Our way structure holds the PFN, combined with the “valid flag” bit, in the structure `pf_n_v`. I told you the name made sense!)

Page frame numbers are addresses shifted right 12 bits. The address has already been shifted right 6 bits, so we just need to shift it another six bits to get the PFN. To turn it into a value suitable for storing in `pf_n_v`, we then shift it left one bit—remember, a way is valid if `pf_n_v`'s lowermost bit is zero.

Now we get the cache line for this index, and check both that line's ways to see whether either one is a) valid and b) maps the referenced address. If so, we increment this procedure's “referenced instruction” counter by the number of instructions in this way.

If not, though, things get interesting. First, we log a miss, and increment the reference counter anyway—after all, the instructions get referenced regardless of whether the cache misses, it just takes longer.

Now we need to figure out which cache way will get filled. The confusing statement `!((pline->w[0].pf_n_v | pline->w[1].pf_n_v) & 1)` basically evaluates as 1 if the bottommost bits of both ways' `pf_n_v` members are 0, or 0 otherwise. That is, if both ways are valid, then we have to choose between them

based on LRF flags. If only one is valid, however, then the non-valid way gets filled.

If both ways are valid, then we XOR the two LRF flags. This is *exactly* how the hardware does it. A way's LRF flag gets inverted whenever that way is filled. The upshot of all this, which you can readily work out for yourself if you wish, is that ways get filled in the following order: 0, then 1, then overwrite 0, then overwrite 1, and so forth.

Next, the `r` bit (the LRF flag) gets XORed against 1, which inverts it as we just saw. Finally, the tag gets stored in the way to indicate that this way is now filled with that particular address.

The data cache uses two functions, one to simulate reads and the other writes.

### **DRead()**

The data cache works like the instruction cache, except that it can be both read and written, and has only 64 lines. So, not `DRead()` works just like `IRef()`, except that it makes provision for writeback events, and it indexes the 64-element `DCache` array.

Another difference is that we have to shift the address right six bits (which is what `SZWAY_LOG2` is defined as) ourselves to get the index, and 12 bits to get the PFN. This is because we get this address from `ATtaCK`'s `ATCK_EFFADDR` dynamic argument, and there's no way to tell `ATtaCK` to shift the value right six bits for us. This will become clear when we look at the instrumentation code.

If either way matches our tag, we chalk up a data read without a miss and continue. Otherwise, we chalk up the read and the miss, and then simulate the cache-fill behavior: Figure out which way is going to be filled, then flip its LRF bit and store the tag. Since this cache is writable, we have to check to see whether the way is dirty. If it is, then we log a writeback event. In either case, it's not dirty anymore, so we clear the dirty bit.

You might be wondering why this function doesn't look more like `IRef()`. Frankly, I wonder that too—there's no actual difference in the result of the two functions, but `IRef()` is written in a clunkier form. I guess it was written first and never got cleaned up.

### **DWrite()**

Fortunately, `DWrite()` looks exactly like `DRead()`, with one exception: On a cache hit, we mark the specified way as dirty, to reflect the results of the write operation.

And that's the cache simulator! Next we'll look at the instrumentation code.

## **Instrumentation**

Most of the instrumentation discussion in `PS2Counter` applies here, too, because the two tools are very similar. They differ in the innermost loops, of course.

The first instruction in each basic block, and every 16<sup>th</sup> instruction thereafter, gets instrumented with a call to `IRef()`. Remember that there are 16 instructions in a cache way. The instrumentation call passes the procedure ID, the instruction's address shifted right six bits, and the number of instructions to load into this line—either 16 or the number remaining in the basic block, whichever is lower.

If an instruction is a load, then it gets instrumented with a call to `DRead()`. If the instruction is a store, then it gets instrumented with a call to `DWrite()`. Both routines take the procedure ID and the dynamic argument `ATCK_EFFADDR`. As discussed, there's no way to shift this argument right six bits here in the instrumentation tool, so we have to let the analysis routine do that (at the cost of a few run-time cycles).

If the instruction has the `CACHE` opcode, it's one of the special EE instructions that manipulate the cache—for instance, this is how cache lines are locked. This version of the tool doesn't simulate cache locking, and truth be told it's an extremely advanced procedure. If you need to work with cache locking, then you understand it well enough to modify this tool yourself!

And that's it for instrumentation.

## Download, Execution and Output

This really does work just like PS2Counter, so refer there for more information. As with that tool, this one only gathers a single frame of data at a time. Unlike that tool, though, the analysis code still has plenty of work to do even when not collecting data—otherwise the cache simulation would become invalid.

Ironically, the cache simulation instrumentation wrecks the application's own cache performance. So what's being simulated is how the cache *would* be behaving, if only there weren't all these function calls every 16 instructions and at every load and store. When you try out this tool, you'll see that it significantly degrades your application's performance. This tool is *not* one that you'll be running all the time!

At any rate, when the user hits `ENTER`, a single frame of data is collected and reported. By now you can figure out what's going on here without my help. The typical output looks like this:



The screenshot shows a Windows Command Prompt window with the title "ps2cache G:\cwu\atck\friday\blow.elf FrameBegins FrameEnds". It displays two tables of cache statistics for the program "blow.elf".

| Procedure           | I-Count       | I-Misses | D-Refs       | D-Misses    | W-Backs     |
|---------------------|---------------|----------|--------------|-------------|-------------|
| SetParticlePosition | 137480        | 0        | 52853        | 5768        | 5209        |
| CreateViewingMatrix | 984           | 0        | 313          | 18          | 17          |
| main                | 569           | 0        | 68           | 8           | 8           |
| SetViewPosition     | 63            | 0        | 20           | 2           | 2           |
| FrameEnds           | 2             | 0        | 0            | 0           | 0           |
| FrameBegins         | 2             | 0        | 0            | 0           | 0           |
| <b>Totals</b>       | <b>139100</b> | <b>0</b> | <b>53254</b> | <b>5796</b> | <b>5236</b> |

| Procedure           | I-Count       | I-Misses | D-Refs       | D-Misses    | W-Backs     |
|---------------------|---------------|----------|--------------|-------------|-------------|
| SetParticlePosition | 137894        | 0        | 52961        | 5768        | 5231        |
| CreateViewingMatrix | 984           | 0        | 313          | 18          | 14          |
| main                | 569           | 0        | 68           | 8           | 7           |
| SetViewPosition     | 63            | 0        | 20           | 2           | 2           |
| FrameEnds           | 2             | 0        | 0            | 0           | 0           |
| FrameBegins         | 2             | 0        | 0            | 0           | 0           |
| <b>Totals</b>       | <b>139514</b> | <b>0</b> | <b>53362</b> | <b>5796</b> | <b>5254</b> |

Fig. 07-03: PS2Cache Output

## Interpretation

This one's simple. "Cache misses are bad, m'kay?" Some cache misses are necessary, but that shouldn't stop you from doing everything in your power to avoid all of them.

You avoid I\$ misses by tightening your code so that your key loops span as few cache lines as possible. In the Blow sample program analyzed above, for instance, you'll note that there are *no* I\$ misses—the entire render loop fits within the cache. So the more compact your code the better, which is just another good reason to move operations out to the vector units rather than handling them in the MIPS core. The other way to really squeeze your code down is to move your tight inner loops into assembly, and use inlining to eliminate branches.

Beyond that, watch for coincidences where your code happens to jump among routines separated by 8K—you'll wind up missing the cache because of the way ways share lines, even if the code in question is smaller than 16K. This is something that a special-purpose ATtaCK tool could detect, even just using static analysis. See "Improvement," below, for a discussion of that idea.

As for D\$ misses, the only way to avoid them is to organize your data or your algorithms in a cache-friendly manner. Let's say you have an AI pathfinding routine that frequently scans your map data, which is stored as a 2D array with rows more than 64 bytes across. Traversing this grid vertically will cause a cache miss every row. Instead, organize the map into chunks, where a particular cell's neighbors are kept within 64 bytes of that cell. Or move the map to scratchpad RAM—that's exactly the kind of thing it's there for.

## Improvement

As mentioned, this tool doesn't simulate cache locking. It's also not smart enough to tell that an address is in the scratchpad and thus won't be cached. The former is hard to fix; the latter is easy—just test the address to see whether it's in the range 0x70000000 to 0x70003FFF. In either case, if you need this feature, you

understand how to add it—ATtaCK is certainly much easier than PlayStation 2 optimization!

As mentioned above, coincidence can cause you to blow your instruction cache inadvertently. It wouldn't be hard to write a static tool to scan for this. For each procedure, determine the range of addresses it calls, shifting them right by six bits and ANDing them with 127 as the cache does. Then look for any function that calls another with overlapping cache lines. Simply by rearranging the order of functions in your C source file, you can probably break up these overlapping pairs.

If you're really seriously using PS2Cache, you need a more detailed output than it currently supports. Right now it gives a good overview of the entire application, at a per-procedure level. A more detailed version would allow you to specify a list of procedures for which to provide an instruction-level profile, so that you know exactly which instructions are blowing the cache. Again, if you need it, then you're advanced enough to write it yourself.

## That's All, Folks!

And with that, we've reached the end of the course. I hope I've given you the confidence and inspiration to write your own ATtaCK tools. These sample tools are nice, and give you a good head start on some common analysis tasks, but you won't tap into the real power of ATtaCK until you start developing custom tools that target your specific code problems.

Thanks for coming, and good luck with your programs!

## How To Contact Metrowerks

|                               |   |
|-------------------------------|---|
| U.S.A. and International      | Metrowerks Corporation<br>9801 Metric Blvd., Suite #100<br>Austin, TX 78758<br>U.S.A. |
| World Wide Web                | <a href="http://www.metrowerks.com/games">http://www.metrowerks.com/games</a>         |
| Games Support Team            | <a href="mailto:ps2_support@metrowerks.com">ps2_support@metrowerks.com</a>            |
| Sales, Marketing, & Licensing | <a href="mailto:games@metrowerks.com">games@metrowerks.com</a>                        |
| Phone                         | 800-377-5416  |
| Fax                           | 512-997-4901  |

# Quiz Answer Key

---

## Quiz Lesson 01

1. The lessons are the most important part of this course. What's the second most important part of the course?
  - B The example programs. Correct. The second most important part of the course is the wide range of sample programs, which demonstrate practical solutions to ATtaCK problems.
2. True or false: Regression testing is the most common method of detecting code errors.
  - B False. Correct. Regression testing is very powerful, but many programs, especially games, aren't suited for it without substantial work. The most common method of detecting code errors is human testing (playtesting). You can develop ATtaCK tools to assist these testers, in addition to writing tools for programmers.
3. What's the difference between using ATtaCK and simply writing analysis code directly into your program, for instance by using asserts?
  - B ATtaCK works with the binary image rather than the source code. Correct. The difference is that ATtaCK can add analysis code to your program's binary executable image, without disrupting your source files or requiring you to recompile.
4. Which team member should develop ATtaCK tools, and why?
  - C A senior programmer, because analysis and optimization are critical. Correct. Code analysis is an important task that deserves the attention of senior programmers.
5. True or false: ATtaCK analysis code is written and compiled just like any other PlayStation 2 program.
  - B False. Correct. ATtaCK analysis code is substantially different from normal PlayStation 2 applications. For example, it uses different project settings, it can't access the normal Sony libraries, and it doesn't have a main() function.

## Quiz Lesson 02

1. How many instructions are in each basic block?
  - D. None of the above. Correct. A basic block consists of a sequence of instructions that do not change the flow of control—in other words, that do not contain any branches or calls. One block could contain anywhere from one to millions of instructions.
2. When ATtaCK iterates through all the procedures in an image, which of the following will appear in the list?
  - C. Both. Correct. ATtaCK searches the program for function calls, but also reads the symbol table. Thus, procedures that are never called show up as well as procedures that do not have symbols.
3. Which of the following is most likely to be a legitimate ATtaCK function?

- B. `atck_iprog_close(atck_iprog_t*)`. Correct. This function begins with `atck_`, has an object name as its second word, and takes a handle to that object as its first argument. It could easily be legitimate—in fact, it is!
4. You have two instruction-object handles, X and Y. Comparing them, you learn that X is less than Y. What do you now know about these two instructions?
- D. X and Y represent two different instructions. Correct. ATtaCK's handle management is completely opaque; there is nothing you can learn from the relative values of two handles. However, if X is less than Y, then X and Y are not equal. Since all handles to the same object will be the same, you know that X and Y are not the same instruction.
5. True or false: You cannot write ATtaCK instrumentation tools in C++.
- B. False. Correct. Instrumentation tools can be written in C++, and indeed there are many advantages to doing so. Analysis code cannot be written in C++.
6. You can use a procedure handle to create an iterator for three of these object types. Which one can not be iterated across in a procedure?
- A. Instructions. Correct. Procedures can create iterators for their basic blocks, entry points and call sites. Only basic blocks can create instruction iterators.
7. True or false: The last instruction in a basic block will always be some kind of branch, call or return.
- B. False. Correct. On the EE, the instruction immediately following a branch executes while the branch is being resolved. The *second-to-last* instruction in a basic block will always be some kind of branch, but the last instruction could be almost anything.
8. Which of the following objects needs to be released after use?
- B. Image. Correct. You only need to release image objects, using `atck_img_release()` or `atck_img_write()`.
9. Of the following, which can appear multiple times within a basic block?
- D. None of the above. Correct. Basic blocks are defined as sequences of instructions that are always executed together. Thus, they are only entered at the beginning, and can only be exited at the end.
10. True or false: Before ending your ATtaCK session, you must free any strings returned from methods like `atck_apname()`.
- B. False. Correct. The strings returned from ATtaCK methods last as long as their parent objects do—for programs, that's until the program is closed, while for everything else that's until the containing image is released. In no case should you try to free these pointers yourself.

### Pop Quiz Lesson 03

```
BeforeProc(06)
BeforeEnt(0600)
BeforeBB(0600)
BeforeInst(060000) / AfterInst(060000)
```

```
BeforeInst(060001) / AfterInst(060001)
BeforeInst(060002) / AfterInst(060002)
AfterBB(0600)
BeforeBB(0601)
```

*Entry point instrumentation only gets called if the procedure is actually entered there.*

```
BeforeInst(060100) / AfterInst(060100)
BeforeInst(060101) / AfterInst(060101)
BeforeInst(060102) / AfterInst(060102)
BeforeInst(060103) / AfterInst(060103)
BeforeInst(060104) / AfterInst(060104)
BeforeInst(060105) / AfterInst(060105)
```

*The call instruction finishes before the call itself takes place.*

```
BeforeInst(060106) / AfterInst(060106)
```

*The instruction in the delay slot also happens before the call.*

```
AfterBB(0601)
```

*A basic-block ends when its last instruction ends, before the call takes place.*

```
BeforeCall(0600)
BeforeProc(1E)
AfterProc(1E)
AfterCall(0600)
BeforeBB(0602)
BeforeInst(060200) / AfterInst(060200)
BeforeInst(060201) / AfterInst(060201)
BeforeInst(060202) / AfterInst(060202)
AfterBB(0602)
AfterProc(06)
```

### Quiz Lesson 03

1. Which of the following is an advantage that sampling has over instrumentation?

- B. Convenience. Correct. Sampling is more convenient, because the target application doesn't need to be modified at all. However, sampling is generally slower, less accurate and less selective.

2. True or false: Instrumentation can be added both before and after entry points.

- B. False. Correct. Instrumentation may only be added before an entry point.

3. True or false: Calls added to the same object execute in the order they were added.
  - A. True. Correct. Calls added to the same *location* through two different objects, on the other hand, execute “bottom up”—for example, a call before an entry point happens before any calls before the basic block that entry point begins.
4. True or false: Instrumentation added after a call site is guaranteed to be executed.
  - B. False. Correct. Some procedures never return to the caller. Use `atck_call_returns()` to check for these procedures.
5. Which of the following lines of code is *not* required in ATtaCK analysis code?
  - B. `#include <atcktargps2.h>`. Correct. This header file contains constants needed on the PlayStation 2. Many analysis routines won’t need these constants at all.
6. True or false: Analysis routines are unable to communicate with the host directly.
  - A. True. Correct. The only way analysis routines can send data to the host is by writing it to memory, which the host can then read.
7. Which of the following is a valid return type for an analysis routine?
  - C. `void`. Correct. Analysis routines are always `void` functions.
8. True or false: You have to be careful when instrumenting branches on the PlayStation 2, to avoid displacing the instruction in the “branch delay” slot.
  - B. False. Correct. ATtaCK handles all implementation details like this for you—you never have to worry about instrumentation code breaking your program.

## Quiz Lesson 04

1. Where can you find the file referred to by `<syscfg>`?
  - C. The directory containing `atck.lib`. Correct. The default system configuration file `syscfg.txt` is found in the `lib\ps2` subdirectory, which also contains the ATtaCK library files.
2. True or false: An ATtaCK tool can monitor the status of the target system without using an event handler.
  - A. True. Correct. While event handlers are probably the best way to monitor the target, you can also poll it using `atck_status()`.
3. True or false: The event-handler arguments *devID* and *progID* must be pointers to structures.
  - B. False. Correct. ATtaCK does nothing with these arguments other than remember them and pass them to the event handler. They can contain any value at all, as long as it can be cast into a `void*`.
4. True or false: The ATtaCK config-file system only lets you read, not write, files.
  - A. True. Correct. However, the config file format is simple text, so you can easily create your own functions to write out these files.

5. True or false: The only way to create a new `iprogram` handle is by calling `atck_iopen()`.
  - B. False. Correct. You also get a new `iprogram` handle back from `atck_finish_write()`.
6. Which of the following is *not* an option expected by `atck_connect()`?
  - D. `devID`. Correct. `devID` is an argument expected by `atck_connect()`, while the rest are options that `atck_connect()` expects to find in the configuration file.
7. Which of the following functions may *not* be called from within an event handler?
  - A. `atck_stop()`. Correct. Event handlers may not call functions that will themselves generate events, such as `atck_wait()` and `atck_stop()`. `atck_kill()` and `atck_continue()`, although they affect the target's execution, do not generate events.
8. When an application has locked a mutex using `atcktarget_lock()`, how can the application be halted or interrupted?
  - C. By the host stopping the target with `atck_kill()`. Correct. When an application has locked a mutex, it may not be halted or interrupted except by `atck_kill()`.
9. True or false: Every member in an ATtaCK structure declaration needs to be named.
  - B. False. Correct. ATtaCK only cares about the structure's layout, and so ATtaCK declarations do not permit you to name members.
10. True or false: You are not required to use the special ATtaCK memory-allocation methods such as `atck_malloc()`.
  - A. True. Correct. These methods exist for your convenience only and are purely optional. However, if you allocate memory with them, you must free that memory with `atck_free()`, not the normal C runtime `free()` function.

## Quiz 05: Designing Analysis Tools

1. True or false: The only purpose of analysis tools is optimization.
  - B. False. Correct. Analysis tools are useful for all aspects of debugging, of which optimization is just one part.
2. True or false: Changing your algorithm is often the best way to fix not only performance problems but bugs of all kinds.
  - A. True. Correct. You should try to fix bugs at as high a level as possible.
3. Which of the following is always feature of a well-designed ATtaCK tool?
  - C. Simplicity. Correct. ATtaCK tools should be focused, lightweight and simple.
4. True or false: The best place to analyze your data is on the target, in the analysis routines.
  - B. False. Correct. Notwithstanding the name, analysis routines should be as streamlined as possible, only gathering the minimum data required. Put the code that actually processes the data on the host.

5. True or false: It's easiest and safest for analysis routines to read registers directly, rather than relying on ATtaCK to pass them in.
- B. False. Correct. Not only is it easier to use ATtaCK to read the registers, but it also ensures that the value doesn't change in between the instrumented instruction and the analysis routine.
6. Which of the following is a valid ATtaCK analysis-routine declaration on the PlayStation 2?
- A. `"GetTarget(valaddr)"` Correct. Keywords such as `regv64` and `valaddr` warn ATtaCK to expect a dynamic argument flag. The instrumentation call itself tells ATtaCK which dynamic arguments to use.
7. Which of the following is a valid place to use `ATCK_EFFADDR`?
- C. `atck_inst_callbefore()`. Correct. `ATCK_EFFADDR` is only valid with instrumentation added before an instruction using `atck_inst_callbefore()`.
8. True or false: You must always use `atck_inst_isallowed()` to verify that a dynamic argument is safe before trying to instrument an instruction with it.
- B. False. Correct. Many dynamic arguments, such as registers, are always safe with instructions and never need to be checked.
9. True or false: An address is all that's required to completely identify any instruction in a program.
- B. False. Correct. Since each image has its own address space, you must have both an address and an image ID to identify an instruction.
10. Which of the following does a MIPS application (that is, a PlayStation 2 game) store on the stack?
- B. Local variables. Correct. Function arguments and return addresses go in special registers, while local variables go on the stack.