SAKTHI: A RETARGETABLE DYNAMIC FRAMEWORK FOR BINARY INSTRUMENTATION

The members of the Committee approve the masters thesis of Amit Vasudevan

Dr. Ramesh Yerraballi _____
Supervising Professor

Dr. David Levine _____

Dr. Mohan Kumar _____

Dean of the Graduate School _____

DEDICATION

Dedicated to OmSakthi the Almighty ,SpiKeY and my family.

SAKTHI: A RETARGETABLE DYNAMIC FRAMEWORK FOR BINARY INSTRUMENTATION

by

AMIT VASUDEVAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2003

ACKNOWLEDGMENTS

I thank OmSakthi (the Almighty) foremost. Without Her I would not be what I am today. I thank SpiKeY, my PC, for being with me all through my life. I love you SpiKe. I thank my family for their support, sacrifice and encouragement. I would like to express my sincere gratitude to my supervising professor and my godfather, Dr. Ramesh Yerraballi, for bringing out my research potential, and for his encouragement and advice during my research. Thanks a ton Dr. Y. I am also thankful to Dr. David Levine and Dr. Mohan Kumar for supporting my work. Last, but not the least, I thank THEM for lending me their hand when I needed it the most. THEY led me into a world of difference.

November 3, 2003

ABSTRACT

SAKTHI: A RETARGETABLE DYNAMIC FRAMEWORK FOR BINARY INSTRUMENTATION

Publication No. _____

Amit Vasudevan, M.S.

The University of Texas at Arlington, 2003

Supervising Professor: Dr. Ramesh Yerraballi

The various contexts in which operating systems and to an extent general purpose applications get used in practice, pose a need to be able to extend certain functionalities to suit the specific context. With access to the appropriate sources, it is very trivial to embed or recast functionality by rebuilding the operating system or application. However, most operating systems and applications are commercial with binary-only releases. This entails a mechanism by which instrumentation has to be achieved regardless of the availability of sources and with minimal system downtime. We propose a "Retargetable Dynamic Framework" for instrumenting constructs at the binary level. It is retargetable in the sense that the framework can be easily adapted to various operating systems and machine architectures. It is dynamic owing to the flexibility of being able to enable or disable the instrumentation at will, capable of capturing dynamic invocations and being able to deploy itself in a newly created or an already existing process. This scheme is much more flexible than executable or dynamic library image manipulation, import redirection or source re-builds.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Operating systems are ever evolving and so are application programs. To gain ground in design and development it is often necessary to instrument and extend existing functionality within the core operating system or applications. With access to the appropriate sources, it is very trivial to embed or recast functionality by rebuilding the operating system or application. However, most operating systems and applications are commercial with binary-only releases. This entails a mechanism by which instrumentation has to be achieved regardless of the availability of sources and with minimal system downtime.

We propose a “Retargetable Dynamic Framework” for instrumenting constructs at the binary level. It is retargetable in the sense that the framework can be easily adapted to various operating systems and machine architectures. It is dynamic owing to the flexibility of being able to enable or disable the instrumentation at will, capable of capturing dynamic invocations and being able to deploy itself in a newly created or an already existing process. This scheme is much more flexible than executable or dynamic library image manipulation, import redirection or source re-builds.

1.1 Instrumentation

Instrumentation is the ability to “control” constructs pertaining to any code. Constructs can either be pure (functions following a standard calling convention) or arbitrary (code blocks composed of instructions not adhering to a standard calling convention). By “control” we mean access to a construct for purposes of possible semantic alteration.

1.2 Motivation

Binary Instrumentation is not a new concept. Techniques of code patching have existed since the dawn of digital computing. There have been various improvements suggested by researchers such as executable image editing, executable and dynamic library import redirection, target function rewriting, dynamic compilation, object wrappers and so on. The next section briefly discusses a number of these works.

Most of these works are tailored towards a specific operating system and machine architecture, employing ad hoc techniques to deploy the instrumentation. Our framework (here onwards referred to as SAKTHI) enables binary instrumentation across multiple operating systems and machine architectures. It is based on target function rewriting and provides a payloading abstraction, which lends itself to implementation on a wide range of commercial and free operating systems. Additionally SAKTHI is capable of providing options for global and local instrumentation, is able to instrument pure or arbitrary constructs and deploy itself in a new or an already executing process with minimal downtime.

SAKTHI has been tested on uniprocessors running Windows 9x, ME, NT, 2K, XP, and Linux. Tests on Solaris, WinCE and VxWorks are currently underway. Consequently, we concentrate on single-processor implementation details throughout the thesis. However, our methodology lends itself to multiprocessor-system implementations as well with certain improvements.

1.3 Types of Instrumentation

Instrumentation comes in various flavors. This section describes the various types of instrumentation possible.

1.4.1 Source Vs Binary

Instrumentation can be applied on application or operating system source code or at the binary level. Source instrumentation is a trivial process and involves inserting programming constructs within the desired region of instrumentation and recompiling the sources to deploy the instrumentation. On the other hand binary instrumentation is a bit more tedious in the sense that one does not have the luxury of studying the sources in order to determine the exact flow and insert the instrumentation. Binary instrumentation thus, works on an instruction level (pertaining to the object code of the target for a specified architecture).

1.4.2 Active Vs Passive

As discussed, instrumentation is the ability to access a construct for possible semantic alteration. Consequently instrumentation comes in two flavors – Active and Passive. Active instrumentation involves altering the semantics of the construct whereas passive instrumentation retains the default semantics while performing other functions specific to the task in hand. For a trivial example, consider instrumenting an operating system call such as the "socket" call to track open socket handles. This would amount to a "Passive" instrumentation, since we are not altering the default behavior of the "socket" call (which is allocate a socket handle), but merely tracking its allocation.

1.4.3 Global Vs Local

Instrumentation can also be categorized into global instrumentation (where all processes see the instrumentation) or local instrumentation (specific to a process or a thread within the process). Again, taking the example of instrumenting the "socket" call. If we wanted the "socket" call to be instrumented in all processes executing or processes that will be executed in the system in the future, we call that Global Instrumentation. On the other

hand, if we wanted one specific process to see our instrumented “socket” call, we would call that a Local Instrumentation.

1.5 Components of Instrumentation

To be able to successfully instrument constructs, we need to make use of the following aids –

1. *Redirection* – The ability to change the flow of a construct so as to enable it to be instrumented.
2. *Payloading* – A mechanism to facilitate Redirection.

As an example, let us once again consider instrumenting the “socket” system call to track open socket handles. Redirection will enable us to replace the existing call with our own while at the same time providing us with an option to invoke the original call from within ours. Payloading will enable us to deploy this Redirection within the context of a specific process or the entire operating system.

CHAPTER 2
BACKGROUND

In the distant past, code patching [1-4, 7] was considered to be a more practical update method than recompiling the entire operating system or application. Some of the works relied on debug symbols of the program being instrumented to get the actual location of that symbol (function) in memory. Other variations included locating the function at runtime and placing a breakpoint instruction at its start in order for the instrumentation engine to get control upon execution. The common approach of all these works was patching the code being instrumented in memory. However, the problem with binary code patching was that the entire patch had to be written in position independent code (delta code) since address relocations were not easy to apply. The other problem with such a method was, finding free executable memory space in the process address space where the code patch was being applied. All in all, binary code patching was a clumsy solution for introducing functionality changes.

Object wrappers were next. These were constructs that were able to alter the semantics of a class or object midstream in execution [5]. However, they were language dependent and not targeted at binary constructs. This meant that the instrumentation engine needed access to the sources of the program or module being instrumented in order to deploy the instrumentation. There were extensions to basic binary code patching to parallel systems [6] using lightweight conditional breakpoints. Multiple processors concurrently ran both the debugger and the target. Shared-memory was used to implement efficient communication between them.

In the Synthesis [8] kernel the concept of runtime code generation to optimize frequently used kernel routines for specific situations was introduced, thereby greatly reducing their execution time. All kernel routines were designed to be polymorphic in nature and could change behavior for performance. Though it was strictly not a general-purpose instrumentation engine, the Synthesis kernel was one of its kinds. However, this was an ad hoc kernel and didn't carry support for existing or future operating systems or applications.

Further research resulted in a class of binary rewriting tools including Atom [9], EEL [12], Etch [15], Morph [16] and the likes. These tools took an executable image and an instrumentation script as their input. The tool then passed over the executable image inserting the instrumentation where applicable using free register discovery mechanisms and supported insertion before or after a basic instruction unit. That is to say, these tools worked at the binary level and needed to know the object format of the executable. They also relied heavily on the underlying machine architecture as they worked on a basic instruction level instrumentation mechanism. The main drawback of these tools was the fact that they were offline methods relying too much on specific executable image format.

During the same time, work in dynamic compilation like VCODE [13] and DCG [14] enabled compilers to generate executable code at run time and replace functionalities. The compiler would introduce certain methods that contained support for dynamically instrumenting selected target functions. These support methods were specific to the language and generally provided as a run time library. The problems with such schemes were that – they were restricted only to applications and dependent on a language.

DyninstAPI [17, 23] was a departure from these systems in that, it allowed for instrumentation to be deployed on running processes as well. It worked by introducing a prologue and an epilogue area for each method that was being instrumented. Again it was limited to applications only. Further, it could not capture dynamic invocations or arbitrary

constructs and did not provide for separate global or local instrumentation abilities. The latest release (version 3.0 as of 2002) of the API however adds capability for arbitrary constructs. The underlying work that made this possible was the dynamic instrumentation technology [10] developed as part of the Paradyn Parallel Performance Tools project [11].

Then there was Detours [20], which introduced the concept of target function rewriting through trampolines. Detours was different from all its predecessors in that it preserved the unmodified construct that could be invoked within the instrumented construct. But, Detours was only implemented on the Windows NT operating system and used specific executable import table rewriting to deploy the instrumentation. Also, there was no support for separate global and local instrumentation or instrumenting active processes. SAKTHI is conceptually similar to that of detours, but addresses some of its shortcomings as mentioned above.

CHAPTER 3
REDIRECTION

Redirection is the ability to change the flow of a construct so as to enable it to be instrumented. This chapter discusses the various methods of redirections possible and the redirection design and implementation for SAKTHI.

3.1 Redirection Methods

There are different redirection methods that were/are used today. This section discusses a vast majority of them.

3.1.1 Call Replacement in Source Code

One of the easiest and earliest methods of redirection is replacing a call to the target construct in the program source, by a call to the instrumented construct. The instrumented construct then performs the intended task and then if need be calls the original target construct to accomplish the default functionality of the target construct. This however entails a program source rebuild as and when new constructs are instrumented. Also, this method relies on the availability of the sources to the program being instrumented.

3.1.2 Call Replacement in Object Code

This scheme is basically a variation of the previous method, the only difference being that the object code of the module or program is disassembled and the instrumented construct is put in place using a raw instruction assembler. This method has one advantage from the previous in that, it doesn't rely on the sources to the program being instrumented. However, this method is static in nature and requires knowledge of the structure of the executable object code being instrumented.

3.1.3 Breakpoint Trapping

This scheme is very similar to the way in which debuggers work. A breakpoint instruction is inserted at the location of the target construct being instrumented. Thus, when the target construct is executed, a breakpoint exception is generated which is captured by the underlying instrumentation engine. The instrumentation engine then performs the required functionality pertaining to that construct and optionally re-inserts the breakpoint insertion to enable instrumentation of future call to the targets. This method works at the binary level and is dynamic. The only drawback to this method is its performance since every call to the instrumented target construct results in a processor breakpoint interrupt being generated.

3.1.4 Import Redirection

This method makes use of the import table (linkage table) in the program object modules to deploy the instrumentation. This scheme can only be applied to shared libraries and dynamic link libraries that export a set of functions to the underlying executable. In this method, the export entries of these libraries are simply remapped to point to their instrumented counterpart. This means that an executable, importing functions off these libraries sees the instrumentation in place. This method works at the binary level and can be static or dynamic depending on the export table implementation of the libraries under various operating systems.

3.1.5 Jump Overwrite/Restore Technique

This is a very simple yet effective technique for redirection. Given below are the steps that come into play when this method is used –

1. Copy "n" bytes at the start of the target construct to be instrumented and store it in a buffer. Where "n" >= length of a jump construct for that particular machine architecture.

2. Insert “jump” construct at the start of the target construct, which jumps to the instrumented construct. For a majority of architectures, this will amount to just placing the jump instruction.
3. During runtime, when the construct is invoked, the instrumented construct gets control that then performs the task at hand. If the instrumented construct needs to call the original unmodified target construct, it will perform steps 4 and 5.
4. Restore "n" bytes from the saved buffer to the memory location of the target construct and call the target construct to perform the default functionality.
5. Perform steps 1 and 2 again.

As seen, this method is fairly straightforward with the only exception of not being able to handle race conditions very smoothly. A race condition occurs when a call to the target construct occurs during the re-insertion of the jump construct. Due to this, such a call may not result in the instrumented construct getting control. What is worse is that, if the call occurs in while the copy is in progress, the system will be in an unstable state.

3.2 Redirection Design in SAKTHI

Our redirection mechanism makes use of the concept of target function rewriting [20]. We use the following terminology to describe the concept of Redirection –

1. *Native Target Function (NTF)* - This is the low-level construct that is to be instrumented (the original function). This construct could have calling conventions like __stdcall__, __fastcall__ or __cdecl__ or, it could be a pure assembly construct.
2. *Instrumented Target Function (ITF)* - This is the user supplied function which does the following –
    - Performs any pre-processing pertaining to the new functionality.
    - Optionally calls the NTF through the Redirector (see below).

- Performs any post-processing pertaining to the new functionality.
- Returns to the caller.

The calling convention of the ITF must match that of the NTF. When instrumenting an arbitrary binary construct that is not a pure function, the ITF must be coded in such a way that it preserves the registers since we cannot rely on calling conventions. It might be argued that this is contrary to the architecture we propose, but recent compilers provide for functions to be coded in a "naked" style with inline assembly using, which, it is trivial to code ITF wrappers for constructs that are not pure functions.

3. *Redirector* - This code is responsible for replacing the NTF by the ITF and for providing a method by which the unmodified NTF can be successfully invoked (optionally) by the ITF.

A construct can be abstracted as a function, expecting zero or more parameters and which does or does not return a value. Based on this fact let us consider a simple construct invocation shown in Figure 2.1(a). Here a source program or module, at some point of its execution "calls" the NTF, which performs its duties and returns. A "call" here refers to an explicit branch instruction or simply the next instruction in sequence, in case of instrumenting arbitrary constructs that are not pure functions. The example uses a hypothetical instruction sequence of a NTF in the Intel 80x86 32-bit format (In reality, it could be in any machine architecture like the Sparc etc.). With instrumentation in place, the construct invocation is as shown in figure 2.1(b). The ITF and the Redirector are collectively called the Redirection code.

The Redirection scheme works on the concept of rewriting target functions in their in-process binary image. In-process rewriting of target functions is a method by which the Redirector is loaded into the target process space employing certain mechanisms (discussed

later). The Redirector then goes ahead and modifies NTFs, which are memory mapped in the process address space. This application of in-process rewriting is crucial to our design and gives our approach an elegant means to deploy the instrumentation into a target process.

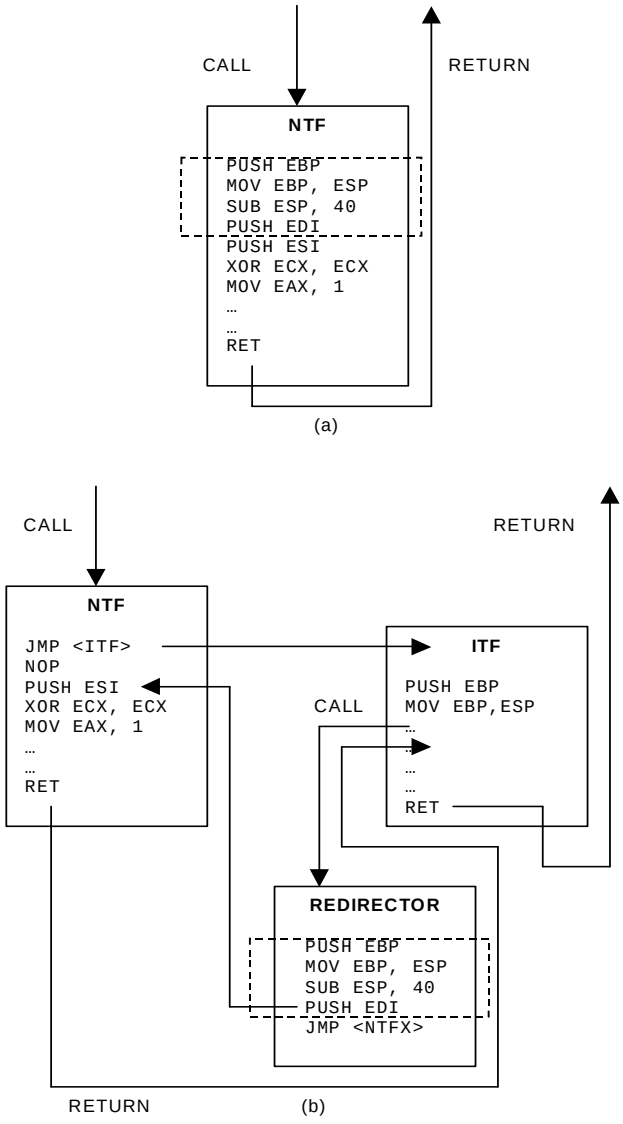

Figure 3.1. (a) Flow of call to NTF without Redirection

(b) Flow of call to NTF with Redirection in place.

The Redirection scheme consists of two steps. The Setup step modifies the NTF to enable transfer of control to the ITF. The Re-Insert step allows for calling the unmodified NTF if needed.

Setup step

Starting at the memory location of the NTF, enough instructions are disassembled so as to fit in a jump construct. These disassembled instructions are then stored in the Redirector. This takes care of not straddling instruction boundaries thereby preserving complete instructions. A "jump" construct is then inserted at the NTF so that it transfers control to the ITF upon invocation. We note that, writing a jump construct comes with its own problems in case of multiprocessors, if it spans cache lines. As our emphasis in this paper is primarily on single-processor systems, we will ignore this issue.

Re-Insert step

When the ITF wants to place a call to the unmodified NTF, it does so through the Redirector. The instructions that were disassembled originally in the setup step are executed and then a jump is made to the following instruction (at NTFX) in the NTF. To the outside world this is logically equivalent to the model presented in figure 2.1(a), as the call is still made to the same NTF and the return from the ITF brings control back to the point from where the call was made as though the NTF had returned.

The Redirection scheme is machine dependent as it generates machine code for the purposes of instrumentation. It makes use of a partial disassembler and an instruction generator to disassemble and insert instructions. However, the framework isolates the machine dependent portions as a back-end module so as to enable various back-ends to be written to support other existing and future architectures.

3.3 Redirection Implementation in SAKTHI

Redirection involves machine specific code such as the insertion of the "jump" construct and the use of a partial disassembler. These vary for different machine architectures. In most cases the "jump" construct directly corresponds to a jump instruction to the target location. However, this might not always be the case. For example, in the Intel 80x86 architecture a jump instruction can refer to any arbitrary 32-bit target location, thereby encompassing the entire machine addressing space. Thus, for the Intel 80x86 the insertion of the jump construct is basically insertion of the machine code for the long jump instruction followed by the target memory location (relative to the long jump instruction) in 32-bits as shown in figure 2.2(a). In the Sparc architecture, there is no support for a single jump instruction to directly reference the entire addressing space. Thus, for the Sparc and other similar architectures, we need to make use of additional instructions as a workaround that allows us to jump to any target location in the machine address space as shown in figure 2.2 (b).

```
JMP <RELATIVE OFFSET OF ITF>
```

(a)

```
SETHI %HI(ABSOLUTE ADDRESS OF ITF), %G5
OR %G5, %LO(ABSOLUTE ADDRESS OF ITF), %O7
JMPL %G0+%G5, %G0
NOP
```

(b)

Figure 3.2. (a) Intel 32-bit 80x86 jump construct and (b) Sparc 32-bit jump construct.

We abstract machine specific code generated at runtime by isolating them in the instruction generator and the disassembler. Instruction generator is a module that is responsible for generation of machine code for a specific instruction. It can be viewed as a partial assembler that generates machine codes for selected instructions. The disassembler is used by the Redirector to disassemble instructions at the start of the NTF, which are then copied to the Redirector code area as previously discussed.

3.4 Comparison of Redirection Methods

As discussed in the previous sections, there are a variety of methods available for redirection. However, most of them are either static or if dynamic, have a performance bottleneck. The most efficient among them is the Target Function Rewriting Technique on which the redirection design of SAKTHI is based. A detailed comparison of various Redirection schemes and their performance is given in [20].

CHAPTER 4
PAYLOADING

Payloading is a mechanism to facilitate redirection. The Redirection code consisting of the ITF and its associated Redirector needs to be in the "target" address space for successful instrumentation. Thus, the Redirection code must be loaded into the "target" address space. Here, "target" refers to either the kernel address space or a user-mode application or library address space. This process is called Payloading.

4.1 Payloading Methods

There are different payloading methods that were/are used today. This section discusses some of them.

4.1.1 Executable Image Modification

In this method, the executable file image is modified to include the payload. This is done by adding extra sections to the executable or by modifying existing sections to incorporate the payload. [9,12,15,16]. As seen, this is a static method that requires the knowledge of the executable image file structure in order to be able to incorporate the payload. Also since the payload is incorporated in binary, it has to be coded in position independent (delta code).

4.1.2 Executable Import Table Rewriting

This method of payloading is borrowed from the import table rewriting technique of redirection. However, in this case, the import table of the executable file is modified rather than the export table of the destination library. This technique adds the Payload to the dynamic library list in the import table whereby the Payload is loaded automatically upon

executable load [20]. Again, this is a static method requiring knowledge about the executable file image being manipulated.

4.1.3 Procedure Linkage Table Redirection

This method is very similar to the previous method in that Procedure Linkage Table (PLT) entries are replaced with exports from the shared library (payload) causing it to be loaded when the process starts. This scheme is applicable to executable image files that rely on the PLT and Global Offset Table (GOT) entries to resolve imports in shared libraries. And example is the Executable Link Format (ELF) found in most flavors of unices. This is also a static method.

4.1.4 Process Memory Space Injection

This technique employs finding free space in the process memory and then writing the Payload out to that and then deploys the Redirectors by hand. This method is dynamic but suffers from one serious drawback in that it fails if there is no free space in the process memory image to accommodate the payload. Also the payload has to be coded in position independent code as the patch is applied in binary.

4.2 Payloading Design in SAKTHI

The payloading mechanism of SAKTHI is dynamic and independent from the executable image file structure. The following sections discuss the payloading design of SAKTHI in more detail.

4.2.1 Payload

The Redirection code along with any associated support functions are collectively called the "Payload". A Payload in its simplest form consists of the following components –

- A one-time initialization procedure.
- A one-time un-initialization (reset or undo) procedure.
- The Redirection code for each NTF to be instrumented.

It is however possible to have other supporting code inside a payload but the above-mentioned are the bare minimum requirements. Let us assume that there is a need to instrument "k" NTFs in the "target" address space. The payload will then contain "k" ITFs each with its redirector as shown in Figure 3.1.

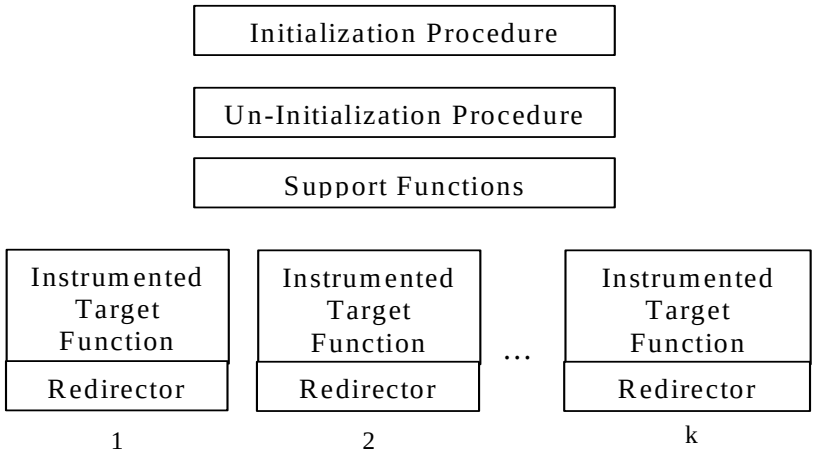

Figure 4.1. Components of a Payload.

The initialization procedure is called only once, when the Payload is loaded in order for it to deploy the instrumentation. This procedure makes use of the Setup step in the Redirectors of the ITFs to deploy the instrumentation. The initialization procedure can also perform other Payload specific initialization such as opening RPC, sockets, IPC connections, message pipes etc. The un-initialization or the reset procedure is called when the Payload is to be unloaded. This procedure restores the NTFs to their original state. It is also responsible for

cleaning up Payload specific resources that were allocated. Thus, the instrumentation is dynamic in that, one could load or unload a payload at will. Further, the framework includes the ability to selectively instrument or un-instrument NTFs at runtime.

4.2.1 Injector

Payloading is performed by an application that we call the Injector. The Injector is a regular application that makes use of the Instrumentation Interface Library (IIL) in order to load the payload. The Injector can load a payload into the kernel mode or a user mode application or library as required.

4.2.2 Payloading Abstraction

The design of a unified Payloading scheme will have to take into account the goals of our framework - (1) to be able to achieve global and local instrumentation (2) to be able to deploy the framework in newly created or already running processes and (3) to be able to implement the scheme on most operating systems and machine architectures. Accordingly, we present our abstraction for Payloading.

Each process has associated with it, two components. The first, its execution context which consists of the machine general purpose registers, floating point registers and other architecture dependent registers and the second its state (ready, running, suspended etc). The steps performed by the Injector are outlined below –

- Step-1: Spawns or attaches to Process-N. The process is now in the "ready" state.
- Step-2: Suspends Process-N.
- Step-3: Captures and saves the current execution context of Process-N (primary thread).
- Step-4: Creates a temporary code area in the address space of Process-N's primary thread (in implementation, this is mostly on the stack of the primary

thread, but for some operating systems, like most flavors of Unix, this might be the code segment) and generates code there to-

- load the Payload.
- call the initialization procedure of the Payload.
- jump to the location in the saved execution context, thereby continuing the normal flow of execution.

- Step-5: Sets up the context of Process-N (primary thread) to start executing code created in the temporary area; Resumes Process- N.

4.3 Payloading Issues

In operating systems that have a single execution space (DOS, VxWorks etc.) the Payload is a "terminate and stay resident" (TSR) executable or a shared library. Our Payloading scheme readily applies here albeit with all of the steps performing very little. The Injector performs step-1 only when a new process is to be launched. Attaching to a running process is trivial as all processes and the kernel share the same address space. Steps 2, 3, and 5 are void. During step-4 the Injector simply loads the Payload as a TSR executable or a shared library.

In operating systems with single-space execution global instrumentation is the default, since all processes including the kernel share the same addressing space. Local instrumentation is achieved by employing a filter on the executable load address and size or on unique process identifiers. That is, the process (es) to which the instrumentation must apply are identified and when one of them is the current running process, the instrumentation is enabled. In single address space operating systems that lack process identifiers, one can trace through the memory control blocks (MCB) and find out the base address where the executable is loaded. Then, when the instrumented function is called, we can filter based on the return address on

the stack by checking to see if it lies within the base and the limit of the MCB of the process executable.

In operating systems that have multiple execution spaces, the Payload takes on various forms depending on the privilege level of the execution space. For kernel space, the Payload is simply a driver or a loadable kernel module. Most operating systems provide a means by which a user-mode application can load and unload privileged code dynamically (device drivers or loadable kernel modules) into the kernel space for execution. The Injector makes use of these services to load the Payload. Only Step-4 of the Payloading scheme applies and the Injector loads the driver or the loadable kernel module during this step.

Operating Systems that do not provide a method, by which privileged code can be dynamically loaded and unloaded, provide a static method for the same. In such cases, the Payload must be listed in the list of statically loaded modules, which are automatically loaded into kernel space at system startup. This would preclude the need for the Injector but would achieve the desired result. However, all of the contemporary operating systems have a dynamic load and unload capability for privileged code. Instrumentation in kernel space is global by default. To achieve local instrumentation, a filter on the unique process identifier is employed as described before.

Payloading in user space is inherently more complex than in kernel space due to two possible implementations of user space, one with Copy-On-Write and the other without. Copy-On-Write is conceptually similar to private copies of memory pages as far as writes are concerned. All the steps mentioned in the Payloading scheme are followed to deploy the instrumentation.

Instrumentation in Non-Copy-On-Write user space is global by default. In Copy-On-Write user space, it is local by default. Local instrumentation in Non-Copy-On-Write user space can be achieved as mentioned earlier by filtering on a unique process identifier.

However, implementing global instrumentation in Copy-On-Write user space is a bit cumbersome. This is done by enumerating all running processes in the system and using the injector to load the Payload into each one of them separately.

We note that if performance is an issue then a kernel level global instrumentation may be a feasible alternative. For example, consider a trivial application that needs to compress a stream of data that is sent over the network on the fly. This can be accomplished by using a global instrumentation at the socket layer in user space or by global instrumentation of the protocol driver in kernel space. However, if the function to be instrumented does not have a kernel level equivalent implementation (as in case of certain general purpose user level libraries), the strategy described here is the only alternative for global instrumentation.

4.4 Payloading Implementation in SAKTHI

The dependency of Payloading on the underlying operating system is greater than that of Redirection on the machine architecture. Payloading in a single address space operating system is straightforward, since all memory is shared among processes and the kernel.

We present the outline of Payloading implementation for Windows and Linux, two popular operating systems. However, it can be extended with ease to most operating systems in wide use today (Solaris, VxWorks, WinCE, and other real time operating systems). The steps outlined correspond to the operating system specific implementation of the conceptual steps already described for the Payloading abstraction in section 4.1.3. A point to be noted is that, though the implementation steps for the operating systems might seem very specific, they are compatible among the same genre of operating systems. For example, the implementation described for Linux is almost the same as for Solaris or any other flavor of Unix. We believe, that for most operating systems our Payloading abstraction will almost always translate to a lightweight operating system specific implementation.

4.4.1 Kernel Space Payloading

Payloading in kernel space (in both Linux and Windows) is equivalent to that in single address space operating systems, since the kernel is a single unit visible to all the processes at all times. Table 3.1 lists the Payload type and shows how step-1 and local instrumentation is implemented in kernel space for various operating systems. The Payload in each operating system is wrapped with the necessary routines that conform to the operating system generic device driver interface specification.

Operating System	Payload type	Payloading steps	Local instrumentation
Windows 9x and ME	Virtual Device Driver (VxD)	Step-1: The injector uses the "CreateFile" call to dynamically load the Payload (VxD).	This is achieved by employing a filter on the process identifier using VMM (Virtual Machine Manager) calls such as "Get_Cur_VM_Handle" and calls to VWIN32 VxD services.
Windows NT, 2K and XP	Kernel Mode Driver (SYS)	Step-1: The injector will first register the driver using "OpenSCManager" and "CreateService" calls. Then, it runs the driver using "OpenService" and "StartService" calls.	This is achieved by employing a filter on the process identifier using KMD support routines such as "PsGetCurrentProcess".
Linux	Loadable Kernel Module (LKM)	Step-1: The injector uses module support functions in the kernel to dynamically load the Payload (LKM).	This is achieved by employing a filter on the process identifier using system calls such as "getpid" which gives the identifier of the currently executing process in the kernel.

Table 4.1. Kernel Space Payloading Implementation for Windows and Linux.

4.4.2 User Space Payloading Under Windows

For the most part, the user space organization of Windows 9x and ME operating systems is very similar to that of NT, 2K and XP. However, Windows 9x and ME user space is non-Copy-On-Write whereas Windows NT, 2K and XP user space is Copy-On-Write. That is, under Windows 9x and ME, the system and shared DLLs are mapped at a shared memory (greater than 0x80000000), which is visible to all user processes across writes. In case of NT, 2K and XP however, the system and shared DLLs are mapped privately in each process space. The Payload in case of Windows 9x and ME is thus a Shared Dynamic Link Library, whereas in case of NT, 2K or XP it is a regular Dynamic Link Library. The Payloading steps are outlined below -

- Step-1: Call "CreateProcess" to create a new Win32 process. In case it is desired to attach to an existing process, the "OpenProcess" call is used instead.
- Step-2: Set the CREATE_SUSPENDED flag during a "CreateProcess" call or if attaching to an existing process, using the "OpenProcess" call and the "OpenThread" and "SuspendThread" calls subsequently. However, in case of windows 9x and ME Win32 subsystem implementation, there is no in built function to obtain the primary thread handle. However, the original "OpenProcess" call under Windows 9x or ME actually has the capability to open remote threads. There is a check performed on the object handle to check whether it's a K32_EPROCESS or a K32_ETHREAD object. The call fails in case it encounters a K32_ETHREAD object. The framework implementation includes a method using which the "OpenProcess" call is made to accept even thread objects.
- Step-3: The "GetThreadContext" call is used for this purpose.

- Step-4: Write the temporary code on the process primary thread stack. Code for calling "LoadLibrary" to load the Payload is directly embedded into the primary thread stack.
- Step-5: Call "SetThreadContext" to restore the primary thread information and the "ResumeThread" call is used to re-start the process primary thread with the Payload loaded in its address space.

Global Instrumentation is the default behavior of the framework under Windows 9x or ME. This is due to the non-Copy-On-Write nature of the user space, making instrumentation visible to all the processes. To achieve global instrumentation under Windows NT, 2K or XP we need to load the Payload in all the existing processes. First, we enumerate the running processes using the "Process32First", "Process32Next", "Thread32First", "Thread32Next" and related calls. Now, for every process the Payloading steps discussed above are performed. Also, the "CreateProcess" call is redirected in every process so that instrumentation can be deployed even on subsequent child processes that might be created.

Local Instrumentation is the default behavior of the framework under Windows NT, 2K or XP. This is due to the Copy-On-Write nature of user space. Hence, when the Redirectors are deployed, the instrumentation is only visible in the process that initiated the deployment. Under Windows 9x and ME, local instrumentation is achieved by employing a filter on the process identifier. The framework uses the "GetCurrentProcessId" call for this purpose. In every Redirector deployed, this call is made use to check whether the process for which instrumentation is desired, is the one that invoked the construct that is instrumented.

4.4.3 User Space Payloading Under Linux

Under Linux, shared libraries wrap all operating system calls. Apart from these shared libraries there are other libraries that are process specific. The user space implementation is Copy-On-Write and the Payload is a shared library. The Payloading steps are outlined below -

- Step-1: Call "fork", to create a new process. In case it is desired to attach to an existing process, the "ptrace" system call is used with PTRACE_ATTACH flag.
- Step-2: To suspend when attaching to an "existing" process is very simple with the "ptrace" system call as it is the operating semantic of the ptrace call. On a new process however, an infinite loop is inserted at the entry point and the process is made to continue its execution. By doing so we render it the status of an "existing" process thereby allowing us to apply the same treatment (use of ptrace) as already described. A point to be noted is that we cannot use the normal "ptrace" on the new process with PTRACE_TRACEME since we want to get control when the dynamic link loader (ld.so, ldlinux.so etc.) has been mapped into the target space.
- Step-3: Use the "ptrace" call with the GET_CONTEXT flag.
- Step-4: Most Unix systems make the stack non-executable, so is the case with Linux. The code at the current instruction pointer is first saved to a buffer. Then the code to load the Payload with "dl_open" call is embedded there in the code segment and finally terminated by an illegal instruction (junk word) that causes an exception. Thus, the Payload loading code is executed and a fault is then reported to the parent (Injector) due to the junk word. The Injector then restores the code buffer and restarts the process at the previous instruction pointer, the only difference being that the Payload is now resident in the target process address space
- Step-5: Use the "ptrace" system call with the PTRACE_DETACH flag set, to let the process continue with the Payload loaded.

To achieve global instrumentation under Linux we need to load the Payload in all the existing processes. First, we enumerate the running processes using the "proc" file-system interface. Then, for every active process, the Payloading steps discussed above are performed. Also, the "fork" and the "exec" calls are redirected (using local instrumentation, the default) in every process so that instrumentation can be deployed even on subsequent child processes that might be created.

If a process calls "fork" to create another child process, then we have no problems as the child and parent share the exact copy of the address space that includes the instrumentation too. However, the situation becomes a bit complex if the child or parent replaces itself with an "exec" call. In this case, the address space is replaced and we need a mechanism by which we can restore the instrumentation. Issuing an implicit "fork" within the instrumented "exec" achieves this. When instrumented "exec" is called, it calls an uninstrumented "fork" to create a new child process. We have two parts after return from the call to the uninstrumented "fork", the parent and the child. The parent part issues a call to real "exec" and never returns. The child waits for some time (to allow for the entry point loop of the executable to be executed), and then does a "ptrace" to attach to the parent process and insert the Payload into it. After that, the child terminates.

Local Instrumentation is the default behavior of the framework under Linux. This is again due to the Copy-On-Write implementation of user space.

4.4 Comparison of Payloading Methods

As discussed in section 4.1, there are a variety of methods for payloading. However, all of them suffer from the fact that, they are either static or depend too much on a particular executable image format. Methods in section 4.1.1, 4.1.2 and 4.1.3 suffer from not being able

to deploy instrumentation on an already executing process. Method of section 4.1.4 success depends upon whether the appropriate executable memory space was found in the target process for writing out the payload. Also in this method, the Payload must be in position independent code or the injector must apply address relocations, which is very cumbersome. Table 4.2 summarizes a comparison of our Payloading method with the ones already discussed in section 4.1.

Payloading Technique	**New Process**	**Running Process**	**Kernel**	**Payload code format**
Executable Image Modification	YES	NO	NO	PIC
Executable Import Table Rewriting	YES	NO	NO	Normal
Procedure Linkage Table Redirection	YES	NO	NO	Normal
Process Memory Space Injection	YES*	YES*	YES	PIC (Normal for Kernel)
Payloading in SAKTHI	YES	YES	YES	Normal

* Not always a success
PIC = Position Independent Code (delta code)

Table 4.2. Comparison of Payloading Methods.

CHAPTER 5
FRAMEWORK

This chapter discusses the organization and performance of SAKTHI. Our framework helps to tie in the mechanisms of Redirection and Payloading such that they can be applied across various operating systems and machine architectures. The framework isolates the operating system and machine architecture dependent parts to provide a consistent interface.

5.1 Organization

The framework organization is as depicted in figure 5.1. Foremost is the Instrumentation Interface Library (IIL) that offers a unified interface, while at the same time concealing the machine and operating system specific modules. In essence, the IIL exposes a set of routines that remain consistent and that are guaranteed to perform the same operation regardless of which machine or operating system the framework is run under. The export routines (function prototypes) are still undergoing significant revision and the current version is detailed in Appendix A. To instrument any code on any OS platform, a user simply invokes routines in the IIL.

The layer beneath the IIL is composed of two blocks - the Machine Architecture Module (MAM) and the Operating System Module (OSM). The MAM interface is designed to contain within it all the machine specific operations including the disassembler and the instruction generator. The MAM also contains the code for ITF wrappers for instrumenting arbitrary constructs. Thus, ports of the framework to other architectures can be done with ease by plugging in a corresponding MAM.

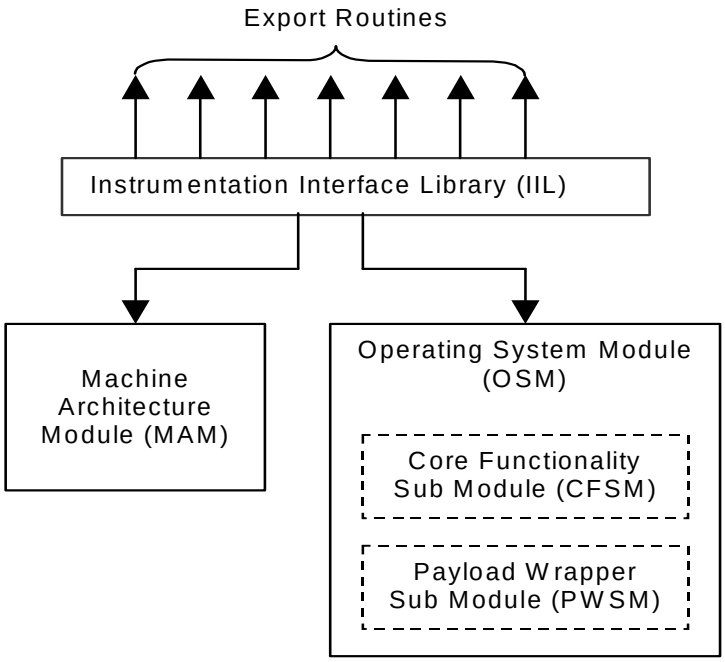

Figure 5.1. Framework Organization

The OSM interface allows for OS specific code dealing with Payloading issues and de-protection and protection of process memory regions. The OSM also is responsible for providing Payload wrappers. By isolating such operations within the OSM, and providing an interface to which the OSM must conform to, it becomes easier to port the framework to various operating systems.

The OSM is composed of 2 sub-modules. They are – (1) the Payload Wrapper Sub-Module (PWSM) and, (2) the Core Functionality Sub-Module (CFSM). The PWSM is responsible for providing a wrapper around the supplied Payload such that it translates to a user or kernel mode Payload for a specific operating system. As an example, consider a Payload for Linux. The user mode wrapper will transform the Payload into a shared library and the kernel mode wrapper into a Loadable Module. Alternatively, for a Payload under

Windows, the user mode wrapper will transform it to a DLL and the kernel mode wrapper will transform it to a Kernel Mode Driver or a Virtual Device Driver. Thus, all that a Payload needs to conform to is the interface specification (load, unload, support services) as discussed. Its actual structure will be decided by the PWSM for that particular OSM.

The CFSM contains operating system specific code for Payloading issues and for the deployment of the instrumentation, which involves interaction with processes and threads such as creating, stopping and restarting them and also with process memory mapping such as de-protection and protection of memory regions. As discussed, each step in the design of the Payload has an operating system specific implementation. The CFSM hides these internal details while providing the functionality desired.

5.2 Performance Evaluation

A quantitative evaluation of our framework involves assessing the contribution of two types of overhead - (1) a one-time Payload load time and Redirector deployment time overhead and, (2) a runtime overhead. The measurements performed under Windows (98SE, ME, 2K) and Linux (kernel 2.4.0) running on a 1.5Mhz AMD Athlon-XP single-processor system, using user space Payloading and local instrumentation, are tabulated in table 4.1. The Payload load time takes about 6-10ms on an average for a new process and 5-7 ms on an average for an active process. Each Redirector deployment incurs an overhead of about 13ms on an average depending on the NTF and the underlying machine architecture. The runtime overhead is minimal and is only due to the "jump" construct that comes to an average of 3-4ms.

Operating System	**Payload load time for new process**	**Payload load time for active process**	**Redirector deploy time**	**Redirector run time overhead**
Windows 98SE	6.230 ms	5.350 ms	11 µs	3µs
Windows ME	7.113 ms	5.637 ms	14 µs	2µs
Windows 2000	8.234 ms	7.223 ms	13 µs	5µs
Linux	9.225 ms	7.533 ms	15 µs	3µs

Table 5.1. Framework Overhead Using User-Space Payloading.

The reader should note that these numbers give an indication of what the equivalent values would be in other scenarios, i.e., user-level global instrumentation and kernel-level instrumentation.

CHAPTER 6
APPLICATIONS

We have successfully applied SAKTHI to a number of real world applications. We discuss a few of them in this chapter followed by a discussion of other scenarios where SAKTHI can find its potential.

The first use of SAKTHI came about in an application involving wireless communication. The system at hand proposed an alternative solution to the default wireless encryption and decryption (WEP) protocol. Though the test-bed had wireless adapters that had an option of disabling WEP, to apply custom transparent encryption and decryption, one had to get into the protocol stack or the physical device chain itself, to intercept packets and perform the encryption and decryption.

The application of the framework to this problem was based on the fact that all the applications use the socket interface to establish and teardown network connections and for sending and receiving data and never directly talk to the protocol stack of the underlying operating system. Moreover, the BSD style socket interface is the de-facto socket API supported by most of the operating systems. The socket APIs were redirected using global instrumentation to go through the encryption/decryption black box. More specifically the instrumented “socket”, “connect”, “bind” and “accept” calls were responsible for establishment of sessions.

Sessions were tracked based on the socket handles. The instrumented “send” and “recv” were the ones performing the actual encryption and decryption on the stream of data. A clean, portable, user level solution was thus found.

As a second application, the functionality of “Vectored Exceptions” that was introduced in Windows XP was extended to other versions of windows like NT/2K. This gives applications desiring the more powerful vectored exception mechanism, a means to achieve binary compatibility among various versions of Windows. We used our framework to redirect “KiUserExceptionDispatcher”, the user mode callback in “NTDLL.DLL” which gets control on an exception.

This handler normally traverses the SEH (Structured Exception Handling) chain to call frame based exception handlers in applications. However, in our redirected “KiUserExceptionDispatcher”, we provide for applications to install their own vectored interrupts which get control before the normal SEH chain is traversed.

Apart from these, the framework finds its use in many other situations. We discuss a few of them below –

1. Enhancing Operating System Capabilities: It is sometimes necessary to enhance or even add certain functionalities in the core OS in order to achieve the required results. Instances may be:
    - To Add Operating System Services: The services provided by the Kernel of the OS can be extended by adding new services. A couple of examples towards this end are discussed below –
        i. Implementing Unicode Variants for Windows 9x/ME: One could dynamically add or remove Unicode Kernel functions that are unavailable in the native windows 9x/ME environments thus making it compatible with their NT/2k counterparts. Also, certain NT/2k Kernel functions don't have an equivalent 9x/ME function. This makes porting applications difficult. With the instrumentation framework, it is possible to provide several equivalent NT/2k functions under 9x/ME thereby

ensuring binary compatibility. The same could be applied to various flavors of Unix OSs.

ii. Support Loading Of Non-Native OS Executable Images: The operating system could be enhanced to load any desired executable image format (ELF, PE, NE, LE etc). This, combined with frameworks to emulate other OS specific calls would provide a method by which executables can be shared among various Operating Systems in its binary form. We call such executables Binary Portable Executables (BPE) and are currently working on an implementation.

- To Implement Operating System Proxy Services: It is possible to extend the Kernel services that are provided by the OS to a distributed environment. Each service behaves essentially like an interface, which could potentially be implemented in conjunction with other workstations. A couple of examples are discussed below –

  i. Software Distributed Shared Memory: Implementing SDSM in systems, which do not have native support, entails the modification of the OS page management services or at a higher level, modifying services provided by the OS towards this end. (signals, structured exception handlers, exception vectors etc.). For e.g., In order to implement SDSM successfully under NT, KiUserExceptionDispatcher can be instrumented in order to provide a first chance exception handling mechanism to the application. Of course the same could be achieved using SEH but is significantly slower.

ii. Distributed File System Implementation: The file operation primitives provided by the operating system can be extended to provide transparent distributed file services.

2. Network Packet Capturing and Filtering: Some operating systems do not have services that enable application/system software to install their callback functions at the network layer. In such cases the instrumentation framework can be used to provide a seamless integration. The underlying capturing/filtering library, which makes use of the instrumentation framework, hides OS specific details while providing a consistent interface.
3. Resource Virtualization and Process Migration: Using this technique we can replace the physical bindings of an application with virtual bindings and use it to migrate active programs between systems as done in the virtualizing operating system [24]. With SAKTHI it is now possible to extend the virtualizing operating system concept to various operating systems and applications.
4. Profiling, Performance Analysis and Monitoring: Application/System software including various portions of the operating system Kernel can be profiled with ease with no access to their sources [8]. Chosen portions of the User or Kernel code can be analyzed and even monitored (both local and remote monitoring is possible) for performance and fault tolerance.
5. Component Re-Architecture and Real Time Systems: Due to the ability of the framework to deploy instrumentation on already running processes, SAKTHI can be used in real time systems where downtime is not desirable. The framework can be used for re-architecture of desired components on the fly with ease and without the need to bring the system down.

6. Debugging and Tracing: With SAKTHI, debugging is made easier at the User and Kernel level and it is possible to obtain a complete trace of each Kernel/library service that an application invokes with the help of the instrumentation framework. It is not required to obtain the sources as operations are done at the binary level.
7. Dynamic Security and Access Control: It is possible to enforce or remove security or access control restrictions at any chosen point in the system during runtime. As a trivial example, one could instrument the connect call in the BSD style socket implementation so that a connect call from a particular host/sub-net to any or a particular port is disallowed or allowed only with authentication. This can be done either at the User level or at the Kernel level.

CHAPTER 7
CONCLUSIONS

This chapter summarizes our contributions, discusses current developments and points towards directions for future work.

5.1 Summary and Current Developments

We introduced a methodology for designing a retargetable dynamic framework, which extends binary instrumentation to various operating systems and machine architectures. We used the concept of target function rewriting for Redirection and devised an abstraction for Payloading that can be readily extended to various operating systems. Furthermore, our framework provides for –

1. separate global and local instrumentation and
2. the ability to deploy itself in a new or an already running process.

The application of our framework will enable existing works related to binary instrumentation achieve portability across different operating systems and machine architectures as well as enable creation of new applications. We are currently testing SAKTHI on VxWorks and Windows CE operating systems to emphasize on its portability across various systems. The framework IIL specification has more or less stabilized in its interface and is presented in Appendix A. We are continuously working on supporting new operating systems and are working on some applications making use of SAKTHI to demonstrate the variety of contexts in which our framework can be applied.

Using SAKTHI, it is easy to produce compelling operating system or application extensions without access to sources or recompiling underlying binary files.

5.2 Future Work

Currently our work has some limitations that are being addressed. Firstly, we need to implement our Payloading abstraction on as many commercial and freely available operating systems as possible to impress its generality. Secondly, our performance results are preliminary in many ways. Our next steps will be to analyze and improve upon our Payloading abstraction to reduce the latency as much as possible. We also need to look at multiprocessor issues. In the near future, we plan to add multiprocessor support to the framework in order to verify our methodology for multiprocessors.

Our Redirection concept also needs to be improved in order to be able to redirect any arbitrary construct. Currently our redirection scheme cannot capture all the variants of arbitrary constructs. As an example consider an arbitrary construct which is well short in size than the unconditional jump construct. This could be a condition in a “for” or a “while” loop. SAKTHI cannot redirect such arbitrary constructs at present. We are working towards a method by which even such constructs can be successfully redirected. This would make SAKTHI support redirection for any construct programmatically possible.

APPENDIX A

INSTRUMENTATION INTERFACE LIBRARY SPECIFICATION v1.0

This appendix lists the interface and the specification of the Instrumentation Interface Library for SAKTHI. The current stable version of the IIL is 1.0. Majority of SAKTHI has been coded in 'C' with parts in assembly where necessary. Hence the details below give the IIL specification in 'C'.

Interface Name: SAKTHIInitialize

Interface Specification:

void SAKTHIInitialize(void)

Interface Description: This interface is responsible for initializing the instrumentation framework. This is the first interface that must be invoked before the framework can be used. This is generally called from the payload initialization procedure.

Input Parameters: None

Result: None

Interface Name: SAKTHISpawnWithPayload

Interface Specification:

BOOL SAKTHISpawnWithPayload(char *targetPath, char *payloadPath)

Interface Description: This interface is responsible for loading the target process executable as indicated by "targetPath" with the payload as specified by "payloadPath". This interface is used when a new process is to be launched with a payload.

Input Parameters:

targetPath = NULL terminated path/filename of the target process executable

payloadPath = NULL terminated path/filename of the payload.

Result:

TRUE = success.

FALSE = error.

Interface Name: SAKTHIAttachWithPayload

Interface Specification:

BOOL SAKTHISpawnWithPayload(DWORD pid, char *payloadPath)

Interface Description: This interface is responsible for attaching to the target process whose process identifier is specified by “pid” and then loading the payload as specified by “payloadPath” into it. This interface is used when a payload is to be launched in an already executing process.

Input Parameters:

pid = unique process identifier of the target process

payloadPath = NULL terminated path/filename of the payload.

Result:

TRUE = success.

FALSE = error.

Interface Name: SAKTHIAllocateRedirector

Interface Specification:

void * SAKTHIAllocateRedirector(void *targetFunction)

Interface Description: This interface allocates a redirector area for the specified "targetFunction" and fills it with the disassembled code at the memory location of "targetFunction". A call to this interface is generally followed by a call to "SAKTHIInstrumentFunction" to instrument the target function.

Input Parameters:

targetFunction = address of the target function for which a redirector has to be allocated.

Result:

Non-NULL address of redirector = success.

NULL = error.

Interface Name: SAKTHIInstrumentFunction

Interface Specification:

BOOL SAKTHIInstrumentFunction(void *targetFunction, void *instrumentedFunction,
void *allocatedRedirector)

Interface Description: This interface is responsible for instrumenting a target function as indicated by "targetFunction" using the allocated redirector and redirecting it to the "instrumentedFunction". The redirector is allocated previously using a call to "SAKTHIAllocateRedirector".

Input Parameters:

targetFunction = address of the target function which needs to be instrumented.

instrumentedFunction = address of the instrumented function.

allocatedRedirector = address of allocated redirector for targetFunction

Result:

TRUE = success.

FALSE = error.

Interface Name: SAKTHIUninstrumentFunction

Interface Specification:

BOOL SAKTHIUninstrumentFunction(void *targetFunction, void *allocatedRedirector)

Interface Description: This interface uninstruments a previously instrumented "targetFunction" using the "allocatedRedirector". This is invoked when the target function needs to be uninstrumented.

Input Parameters:

targetFunction = address of the target function that needs to be uninstrumented.

allocatedRedirector = address of the allocated redirector for the target function.

Result:

TRUE = success.

FALSE = error.

BIBLIOGRAPHY

[1] Stockham, T.G. and J.B. Dennis. FLIT- Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0.*Department of Electical Engineering, MIT, Cambridge, MA,Memo* 5001-23, July 1960.

[2] Evans, Thomas G. and D. Lucille Darley. DEBUG – An Extension to Current Online Debugging Techniques. *Communications of the ACM*, 8(5), pp. 321-326, May 1965.

[3] Digital Equipment Corporation. *DDT Reference Manual*, 1972.

[4] Dan G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23, September 1988.

[5] Dan G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23, September 1988.

[6] Aral, Ziya, Illya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87-95. Boston, MA, April 1989.

[7] Kessler, Peter. Fast Breakpoints: Design and Implementation.*Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 78-84. White Plains, NY, June 1990.

[8] Henry Massalin, Calton Pu: Reimplementing the Synthesis Kernel.*USENIX Workshop on Microkernels and Other Kernel Architectures* 1992: 177-186

[9] Srivastava, Amitabh and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196-205. Orlando, FL, June 1994.

[10] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., pp. 841-850.

[11] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer*, 28(11), 1995, pp. 37-46.

[12] Larus, James R. and Eric Schnarr. EEL: Machine-Independent Executable Editing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291-300. La Jolla, CA, June 1995.

[13] Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. Laboratory for Computer Science, MIT. *Appeared in SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, Philadelphia, PA, May 1996.

[14] Engler, Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System MIT, University of Arizona.

[15] Romer, Ted, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the USENIX Windows NT Workshop 1997*, pp. 1-7. Seattle, WA, August 1997. USENIX.

[16] Zhang, Xiaolan, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automated Profiling and Optimization. *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*. Saint-Malo, France, October 1997.

[17] Hollingsworth, Jeffrey K. and Bryan Buck. *DyninstAPI Programmer's Guide, Release 1.2*. Computer Science Department, University of Maryland, College Park, MD, September 1998.

[18] Galen Hunt and Michael Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pp. 187-200. New Orleans, LA, February 1999. USENIX.

[19] Hunt, Galen C. and Michal L. Scott, "Intercepting and Instrumenting COM Applications", 5th *USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, San Diego, California, USA, May 3–7, 1999

[20] Galen Hunt and Doug Brubacher, Detours: Binary Interception of Win32 Functions, *Proceedings of the 3rdUSENIX Windows NT Symposium*, July 1999.

[21] Tom Boyd and Partha Dasgupta, Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization, in the *2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. 3, June 2000, pp. 1431-1437.

[22] Shu Zhang, Mujtaba Khambatti and Partha Dasgupta, Process Migration through Virtualization in a Computing Community, *13th IASTED International Conference on Parallel and Distributed Computing Systems*, August 2001.

[23] Hollingsworth, Jeffrey K. and Bryan Buck. *DyninstAPI Programmer's Guide, Release 3.0*. Computer Science Department, University of Maryland, College Park, MD, January 2002.

[24] Tom Boyd and Partha Dasgupta, Preemptive Module Replacement Using the Virtualizing Operating System, *Workshop on Self-Healing, Adaptive and Self-MANaged Systems (SHAMAN)*, New York, June 2002.

BIOGRAPHICAL INFORMATION

Amit Vasudevan received his Bachelor's Degree in Computer Science and Engineering from the B. M. Sreenivasaiah College of Engineering (Bangalore University), Bangalore, Karnataka, India in 2001. He began his graduate studies in Computer Science and Engineering at The University of Texas at Arlington in August 2002 and received his Master of Science degree in December 2003.

His research areas are operating systems, system software, code re-engineering and debugging. He loves to explore the internals of all operating systems and applications or libraries and document its structure and undocumented interfaces. He believes that every piece of software in this world should be free of charge and open source.