# SIND: A Framework for Binary Translation

Trek Palmer Dino Dai Zovi Darko Stefanovic

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
e-mail: {*tpalmer,ghandi,darko*}*@cs.unm.edu*

December, 2001

**Abstract**

Recent work with dynamic optimization in platform independent, virtual machine based languages such as Java has sparked interest in the possibility of applying similar techniques to arbitrary compiled binary programs. Systems such as Dynamo, DAISY, and FX!32 exploit dynamic optimization techniques to improve performance of native or foreign architecture binaries. However, research in this area is complicated by the lack of openly licensed, freely available, and platform-independent experimental frameworks. SIND aims to fill this void by providing a easily-extensible and flexible framework for research and development of applications and techniques of binary translation. Current research focuses are dynamic optimization of running binaries and dynamic security augmentation and integrity assurance.

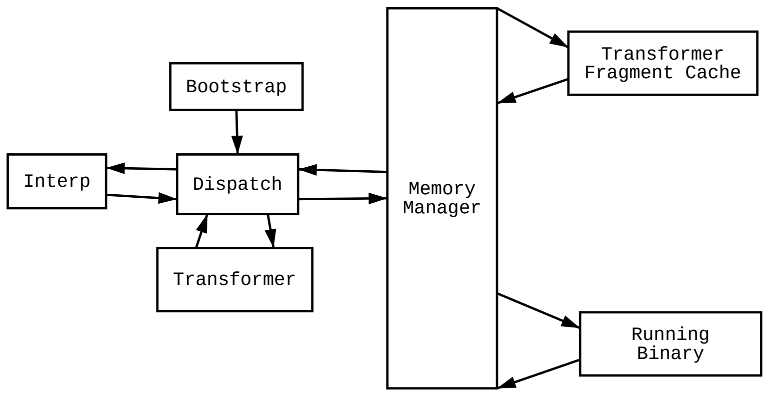
## 1 Introduction

The ideas of program optimization and instruction translation are not new, however their primary application has been in the static process of compilation. In recent years, efforts have been made to adapt these concepts to a dynamic setting. Projects such as Dynamo [2] and the Java HotSpot [12] system attempt to overcome (in a dynamic context) some of the barriers to static optimization such as those that object-oriented languages can create. The basic idea is that while a compiler must treat every code branch as equally possible, a dynamic translator can let the program itself tell the translator which branches are important and likely. This information can be used to transform the running program into a more efficient version. It can also be used to more intelligently monitor its execution for potentially dangerous activity.

SIND is a modular framework for dynamic program profiling and transformation. SIND emerged when we were searching for an existing dynamic optimizer and found either non-free implementations or optimizers tied down to one specific architecture. We then decided that what was needed was a free and platform independent dynamic translation framework that could be used by everyone doing dynamic binary translation research. By examining other dynamic optimizers, a basic structure was abstracted. This structure (detailed in the design section), would allow for multiple underlying and target architectures to be supported, as well as for multiple translation and profiling tasks. Specifically, the developers wanted to ensure that SIND supported both the SPARC and PowerPC architectures and could be used for both dynamic optimization and runtime security checking.

## 2 Design

The SIND framework is composed of a group of inter-operating modules. The bootstrapper loads and halts the running binary just before execution of the mainline (but after library linking). Then control is handed over to the dispatcher which is the central communications hub of the SIND process. The dispatcher mediates inter-module communication, and provides interfaces to allow seamless module-swapping. The dispatcher coordinates with the memory manager to allow transparent and safe reads and writes between the running binary and the separate SIND process' address spaces. After initializing the memory manager the dispatcher starts up the interpreter which begins software interpretation of the running executable. This allows non-invasive binary profiling and allows the interpreter to gather statistics and track behavior without having to modify the source binary at all. When an 'interesting' code trace is encountered, the interpreter can do one of several things. It can immediately 'bail out' and halt execution (if, for instance, an unsafe operation occurred while the binary was executing). It can start a code trace and fill an instruction buffer with all subsequent instructions (until an end-of-trace condition is satisfied). It can also modify 'unsafe' input data to guarantee 'safe' behavior of the running binary (such as truncating a buffer to prevent it from overwriting stack frame boundaries). If a code trace was generated, the interpreter can then hand the instruction buffer (along with processor state information) to the dispatcher for transformation. The transformer module is a program that takes code traces as input, safely transforms them into an equivalent piece of code and then requests that the dispatcher place them in a code fragment cache. This cache is then later used in place of straight interpretation when that particular code trace is encountered again. Such code fragments may then be persistently stored so that on subsequent executions of the application, the cache can be pre-filled. Thus, we eliminate costly re-transformations of the same code. The following diagram illustrates the module relationships:



## 2.1 Dispatcher and Bootstrapper

The system bootstraps by using the debugging facilities of the operating system to run the target executable as an inferior process. The two current experimental implementations use the `ptrace` [11] and `procfs` [10] facilities of the Solaris operating system. The inferior process is halted at an appropriate point (usually the beginning of execution, `_start`, or upon entering the `main` function) and the process is resumed in a software interpreter.

The Dispatcher is primarily concerned with correctly initializing other SIND modules and in mediating communication between them to remove unnecessary inter-dependence. After the bootstrapper has successfully started and halted the executable process, the dispatcher will initialize the memory manager to allow controlled reading and writing from the address space of the binary to the SIND address space. It will then initialize the interpreter with references to the appropriate memory manager. This memory manager will handle all the details of accessing the running program's memory and so will present the interpreter with a simple interface allowing reads and writes. In the event that the interpreter generates a code trace that requires transformation the dispatcher will initialize an appropriate transformer and its cache, and create a link between them through the memory manager. This link will then later be used when the interpreter encounters a previously transformed code trace. A specific advantage of the dispatcher system is that it allows multiple modules of the same type to co-exist without having to explicitly know about each other. This can be advantageous in situations where the specific modules need to be swapped or duplicated while SIND is running.

## 2.2 Interpreter

The SIND interpreter is a software Instruction Set interpreter. The interpreter maintains register and processor state in SIND memory as well as further useful information about the running binary. Because the interpreter is for a specific Instruction Set and not for any specific processor implementation, the interpreter doesn't need to worry about low-level hardware issues such as device control. However, the interpreter must be completely aware of instructions that may transfer control out of SIND. The interpreter must guarantee consistency between its registers and the underlying hardware state when control is restored. In current experimental implementations the SPARC and PowerPC interpreters interpret only user-mode instructions and let the OS handle supervisor code, updating registers after returning from system calls. This is not to suggest that supervisor code could not also be interpreted, but such implementations would be considerably more complicated [9] [7]. As the executable is interpreted, the interpreter gathers statistics and profiling information. This information may simply be archived for later analysis; or, more interestingly, may be used to trigger actions by the interpreter.

The Interpreter may simply halt the running binary based on the run-time statistics. This could be useful for a number of applications (such as preventing a potentially unsafe code segment from executing). The interpreter may also modify 'unsafe' input in order to prevent unsafe execution. This could be used to stop an overflowing memory buffer from corrupting the stack (by truncation), among other things. The interpreter may also heuristically determine that a specific code trace is 'interesting'. This could, for instance, take the form of an often-executed function call or branch (indicative of a loop). The interpreter may then start filling an instruction buffer with the subsequently executed code. While the interpreter is in this trace mode it will associate auxiliary information with specific instructions to aid any possible transformation (such as the destinations of indirect branches or function calls). The trace is stopped when the interpreter encounters an end-of-trace condition (such as another start-of-trace condition, or some constant maximum limit on trace size). The interpreter can then hand the trace over to the dispatcher for transformation and caching. Then, as long as the trace remains valid (determined by hit-or-miss statistics gathered by the fragment cache) the interpreter will transfer control over to the cache the next time that trace is encountered.

## 2.3 Memory Management

The memory manager module coordinates memory reads and writes between the SIND, Running Executable, and Fragment cache address spaces. This should allow platform-independent interfacing with multiple address spaces. This also permits utilization of more efficient platform-specific optimizations without exposing the details of such optimizations to modules on the other side of the interface. Debugging interfaces such as `ptrace` and `procfs` allow access to an inferior process' address space (albeit with the expense of a system call for each operation). Future optimizations include the use of shared memory pages and other forms of shared memory.

## 2.4 Transformers and Fragment Caches

A SIND transformer module is responsible for transforming a code trace gathered by the interpreter, and placing the transformed version in a cache for later use. Such a transformer could, for instance, apply several linear-pass optimizers to the trace to speed-up subsequent executions. Or it could place guard code around potentially unsafe code blocks to automatically perform safety checking. The fully transformed trace is called a fragment, and is placed in the fragment cache for direct execution by the Interpreter/Dispatch.

The fragment cache is responsible for maintaining an efficiently accessible list of fragments, as well as guaranteeing that fragments correctly return control back to the SIND process. This is done by adding entry and exit references to the fragment so that a prologue function is called before entry of the fragment proper, and every exit point of the fragment jumps to an epilogue function which correctly restores control and state to SIND. The fragment cache may further support 'linking', by which fragments that branch to each other are linked together into a super-fragment, which eliminates the overhead of having to leave the fragment cache and then immediately re-enter it.

## 2.5 Offline Processes and Persistence

Because SIND is likely to generate similar fragment caches from one execution of a given application to the next, it makes sense to store the fragment cache in some persistent fashion so that the next time the application is run through SIND, the fragment cache can be pre-filled to eliminate costly re-interpretation. Persistent code fragments also allow for offline processing of the fragments. Such processing would normally be far too expensive to do at runtime and so could be started at some later point to work on the persistent fragments. Examples include expensive optimization routines, code verification, or generation of proof carrying code from the fragments.

# 3 Applications of the SIND Framework

The SIND Framework is meant to be a flexible binary translation software suite and so must accommodate a variety of possible applications. Several such uses for the SIND framework are presented below.

## 3.1 Dynamic Binary Optimization

A popular use for binary translation is to exploit run-time properties of programs to optimize them even further that static compilation can. For instance untaken branches can be eliminated, function calls can be inlined, and indirect branches can often be converted into direct branches. In such a scenario the interpreter would be for the same architecture as the one the SIND process was running on. It would collect traces of commonly executed 'hot code', and the transformer would be a collection of simple linear-pass optimization routines to speed up the trace. The offline processing of the fragments would then be much more intense optimization routines to generate even more optimized code fragments.

This is the current focus of the experimental implementation for the SPARC architecture. Building upon the work of the Dynamo and FX!32 [4] projects, the SIND Dynamic Optimizer hopes to combine efficient binary optimization with intelligent persistent storage without being restriced to a specific platform.

## 3.2 Runtime Assertions of Process Integrity

The low-level interpretation and instrumentation offered by SIND makes possible the dynamic instrumentation of the process to ensure integrity and security. In particular, it allows SIND to protect the inferior process from many sorts of buffer overflows. By taking special precautions with respect to the saved program counter in the stack frame, classic "stack-smashing" [13] attacks can be detected and the process can be halted. This can be performed through the interpretation and instrumentation of the code traces. Because this checking can happen "out-of-band" in the SIND process, it is very difficult to bypass by an attacker. Similar protections can be extended to all saved registers in the stack frames, limiting the effects of a stack buffer overflow to other automatic variables on the stack.

## 3.3 Transparent Binary Instrumentation and Profiling

Because the SIND interpreter interface allows any profiling information to be gathered, it allows for custom interpreters to be constructed that will gather statistics on running programs for research purposes. Apart from the standard timing information SIND could gather information about specific segments of code. Certain chunks of the binary could be flagged as interesting or ignorable, leading to a high-resolution timing system. For instance, GUI programs could have the wait portion of their event loop ignored, thus allowing more accurate timing of the program's logic sections. Or more interactive benchmarks could be used to analyze program execution patterns.

### 3.4 Dynamic Binary Translation

Because the SIND dispatcher isolates modules from one another, a dynamic binary translator could consist of an interpreter that translates code from one architecture into instructions on another, with a transformer then used to optimize the translated fragments. In this way, code for two similar platforms (Solaris/SPARC and Solaris/x86, for instance) could be executed on either platform, with one interpreter and transformer being used for native code and another interpreter transformer pair being used to translate foreign binaries.

## 4 Dynamic Binary Optimization

The SIND framework is designed to facilitate implementation of a dynamic optimizer, and thus includes all the necessary parts for active transparent profiling and optimization. As in many binary translators [2] [7] [9] [4], native code is executed in an interpreter for the purpose of transparently gathering information. Specifically, a heuristic analyzes the current instruction and determine if it is "interesting", and if so it associates a counter with that particular instruction. Each time that instruction is later encountered, that counter is incremented. When this counter exceedes a given threshold, the interpreter will begin to gather a trace. A trace is a sequence of instructions from one "interesting" instruction (the trace head) to the next "interesting" instruction encountered (the trace tail). Note that it is possible for the head and tail to be the same (as in a loop). Also note that a trace may include function invocations. Once a trace is identified, this trace is then combined with some auxiliary information (such as the value of a given register at the time an indirect branch/call was taken) and is then handed over to dispatch for processing by a transformer. The transformer optimizes the trace and places the optimized version (now called a 'fragment' in Dynamo parlance) in a cache. Then the next time the interpreter encounters that trace head instruction, control then transfers to the cache and the optimized version is run on the processor directly. Based on the idea that most of a program's execution is confined to a minimal subset of the code, after an initial warm-up period most of the execution will actually take place within the trace cache. This more than makes up for the initial overhead of running an interpreter.

### 4.1 The Optimizing Interpreter

The interpreter does no actual optimizing itself, although pains should be taken to make it as efficient as possible. Of primary concern in the interpreter is both choosing "interesting" instructions, and picking an appropriate threshold value. Currently most dynamic optimizers fixate upon backwards-taken branches (because these are indicative of loops), and so consider any such frequently taken branch "interesting". This has the advantage of being both intuitive and simple to implement. It is also a relatively fast heuristic. However, as SIND makes no requirements on what internal profiling the interpreter performs, different heuristics could be easily tested and compared on sample code, perhaps resulting in a superior heuristic.

Likewise, the threshold value can also be determined by experiment. Current dynamic optimizers seem to pick values of about 15 [2] [1], but little has actually been written on why that value seems best. With SIND it would be relatively simple to create a series of Interpreters with different heuristics and thresholds, and to subject them all to a battery of tests to determine the best heuristic/threshold pairing for a specific platform. Unlike Dynamo or DAISY, SIND's flexible module framework would allow several different interpreters to run independantly, each with their own heuristics and thresholds.

## 4.2 The Optimizing Transformer and Trace Cache

The Transformer and Trace Cache are where most of the work takes place. The transformer must take a trace and then transform it into a functionally equivalent, optimized version. Time constraints on the Dynamic Optimizer effectively restrict the sorts of transformations that can be done to linear pass optimizations. Therefore the more complicated optimizations often found in compilers cannot be used here. But, more hardware specific considerations can be used. For instance, translating indirect branches into direct branches to eliminate an unnecessary memory reference. Also, because the trace crosses function boundaries, one essentially gets inlining for free. Only minimal translation needs to occur to remove the unnecessary function call and return.

The transformer also has the advantage that the trace's scope is significantly narrowed from that of the whole program. This means that certain optimizations that could not be performed on the whole program may be valid in the trace's restricted context. For instance, a variable that may vary over the whole program's execution may be locally constant in the context of the trace, and so may be folded away. Once such transformations have been performed, the fragment is then handed over to the fragment cache for future execution.

The fragment cache has two important responsibilities. It must guarantee that any executing fragment will return control to the cache (and therefore SIND) after executing, the cache must also monitor the usage of loaded fragments and be able to remove stale fragments. Control over the fragments is maintained by examining each submitted fragment and cataloging each possible exit point from the fragment. Special prologue and epilogue code blocks are then created to verify that the fragment was called correctly and to initialize a context switch back to the fragment cache. The target addresses of the fragment's exiting branches is modified to point to the fragment epilogue. Because a loaded fragment cannot be entered except through a call to its prologue, and cannot be exited except by a call to the epilogue, the fragment cache guarantees that no fragment will "break out" of SIND (this was a potential problem with Dynamo's memory layout). The cache has several options for monitoring hit-or-miss statistics. In the Dynamo project, a fairly course-grained measure was used, and the whole cache flushed when its contents were deemed too cold. In SIND, the thought is to have finer control by measuring statistics separately for each fragment. This would then eliminate the costly dump-everything-and-reload-cache procedure, and spread the cost over the whole execution in an incremental fashion. Although SIND is not yet in a state to support such experiments, our goal is to develop SIND so that this measurement can be taken. Optimizations to the basic fragment cache are possible, costly fragment cache exits and entrances may be eliminated by internally linking together fragments. In practice, this has greatly increased execution speed of the optimized binary [9] [2] [7] [12].

## 4.3 Persistence and Offline Processing

Unlike most other dynamic optimizers [2] [7] [9], the SIND framework facilitates the addition of further components to the dynamic optimizer without having to rework pre-existing modules.

With this in mind, it should be easy to integrate a more permanent cache “archiver” module with the existing runtime dynamic optimizer. This archiver would make persistent dumps of the fragment cache to disk, allowing any future execution of the binary to pre-load the fragment cache, and thus eliminate costly interpretation/optimization calls. Onto this system it should be simple to graft an offline batch optimizer. Such an optimizer would run in the background and perform expensive optimizations on the archived fragment caches. This would result in faster fragments [4], and wouldn’t impact the running time of the dynamic optimizer.

### 4.4 The Memory Manager

The primary function of the memory manager is to maintain the integrity of the virtual memory space of a process both when the process is executing within the software interpreter and when it is executing natively on the processor. There are several approaches to this. Some include translating memory references when running in the interpreter to access the values in the address space of a process. Another approach involves mapping pages between the SIND process and the inferior process such that the memory is accessible in both. A third approach is to run SIND and the target process in one address space so that memory accesses do not need to be translated nor relayed when the target process is being interpreted [2].

The goal of the memory manager is to maintain this consistency without further impacting performance. Methods of accessing the address space of another process typically require a system call. Although this may be mitigated through batch changes (when switching back to native execution, for example), the cost of a mode switch may be too high. Higher-performance but more platform-specific options such as shared pages may exist and be preferred. Some platform-specific methods may require specific operating system facilities or even modifications to the kernel image through loadable modules. Through the abstractions provided by the SIND architecture, portable and implementation-dependent memory managers will be written such that the fastest supported method is used. This ability to dynamically select a memory manager provides a level of flexibility lacking from many other dynamic optimizers.

## 5 Runtime Assertion of Process Integrity

The low-level control of a binary translation system makes it possible to externally augment the security of an executable. In particular, process integrity can be asserted through defenses against code injection attacks. Code injection attacks, such as buffer overflows and format string attacks, subvert the vulnerable process by taking control of the process and directing it to execute injected code. Typically, the first step in this is overwriting control structures such as function return addresses and Global Offset Table entries. Through software interpretation and code instrumentation, many of these structures can be protected.

### 5.1 Protecting Return Addresses

The typical target of “stack-smashing” buffer overflow attacks is the function return address stored in the stack frame. Controlling this value can cause the vulnerable program to return control into dynamically injected code. Several protections against this involve the use of compiler extensions to check the validity of a saved return address before jumping to it [5]. However, this can also be done dynamically.

Some static stack protections like StackGuard alter the stack frame format to include a “canary” value before the return address. These solutions require a special stack frame format and hence require recompilations of the executable. In a StackGuard protected executable, before returning from a function, this canary is checked to ensure that it hasn’t been overwritten (indicating that the return address has been overwritten also). If the canary does not match its original, random value, execution of the process is halted. With knowledge that such a system is in use, an attacker may use other attack techniques to overwrite a saved return address exactly (without altering the canary value) [3]. More recent versions of StackGuard XOR the return address with the canary to detect if either has changed. SIND, however, is not vulnerable to this method of attack. Regardless of whether an attacker knows that a system is running in SIND, special values such as return addresses are verified with values in a separate address space, and are therefore inaccessible to an attacker.

When running in a software interpreter, the return address of a function can be duplicated out-of-band in the interpreter. Upon returning from a function, this value can be checked against the address stored in the function linkage. If there is a discrepancy, the process can be terminated, or SIND may attempt to correct the return address so that the process may continue executing. However, precautions must be taken so that the continued execution is safe. When running natively on the processor, the code may be instrumented dynamically to add similar integrity checks. For example, on the SPARC architecture, instead of restoring the registers in the branch-delay slot of the `ret` instruction, SIND may translate the instructions into a sequence that restores the registers, compares the return address to an immediate value, and then jumps to the calling function.

## 6 Transparent Binary Instrumentation and Profiling

SIND already includes the necessary components for transparent profiling and instrumentation as these are needed for both dynamic optimization and process integrity assertion. So a pure profiler would simply be a custom interpreter that would report on the behavior being profiled. Instrumentation can be achieved with a custom transformer. Note that this instrumentation is not directly added to the binary itself, and so is transparent. SIND’s flexibility allows for a combination of optimization with instrumentation, which may result in instrumented running times close to uninstrumented times.

## 7 Dynamic Binary Translation

Dynamic translation is the rewriting of a binary compiled for one platform to another target platform. One can see that with a few modifications, the Dynamic Optimizer can be turned into a translator. Simple translation of instructions and simple system call translation would easily allow a binary compiled for Solaris/SPARC to run on Solaris/x86. Even different platforms could be supported (as in FreeBSD [8] and Solaris Linux emulation [6]). As in dynamic optimization, binaries would be translated on the fly, but offline persistence and optimization would become more important. The persistently stored fragment cache dumps would then become the translated binary on the target system. As in the FX!32 project, offline processing of the cache dumps would mean that after a few run-throughs, a foreign binary would be executing at near-native speeds.

## 8 Related Work

StackGuard [5] uses static compiler extensions to augment programs with increased stack protection. By placing "canary" values before return addresses in stack frames, stack overflows are detected when the canary is altered. Later revisions correlate the return address with the canary so that the alteration of the return address is detected even if the canary is not touched.

Dynamo [2] is an experimental dynamic optimization system for the HPPA architecture. It used both interpretation and simple optimization to speed up the execution of native code. The fragments were stored in a cache and internally linked, however the cache contents were never stored persistently and using Dynamo required an HPPA workstation running a version of HP-UX with a modified linker.

FX!32 [4] is a program for running WinNT binaries compiled on an x86 on an Alpha running the same operating system. FX!32 translated the binaries into x86 instructions, and later "batch-processed" the translated fragments into heavily optimized versions. FX!32 achieved good runtime performance exclusively through offline batch optimization.

Both the FreeBSD [8] and Solaris [6] operating systems have support for runtime Linux emulation. No binary translation occurs, so this only works on x86 platforms. The emulation consists of system call translation, and so only works on binaries that don't rely upon Linux-specific features (such as direct access to video hardware).

The Transmeta Crusoe [9] processor uses "code-morphing" techniques to translate foreign instructions into the Crusoe's native VLIW instruction set. This "code-morphing" software exists at a higher level than standard processor optimizations such as branch prediction. This software relies upon dynamic optimization techniques in order to compensate for the Crusoe's lower clock speed.

The Java Hotspot VM [12] is a Java virtual machine that uses both JIT compilation and runtime optimization to greatly speed execution of running Java bytecode. However, many of the HotSpot optimizations are very specific to the Java language.

The DAISY binary translator [7] is a dynamic optimizer for the PowerPC architecture. Like Crusoe, it sits above the bare processor but below the operating system. Although now open source, DAISY requires extra hardware to effectively function.

## 9 Future Directions

### 9.1 OS Interfaces

Further development of the SIND Dynamic Optimizer might include moving SIND logic into kernel-space (for instance, as a loadable module). This would allow for much more efficient access to the running binaries' address space, as well as more efficient transfer of control to other kernel functions.

## 9.2 Hardware ‘Aware’ Transformers

Another direction for development might be the creation of very hardware specific interpreter/transformer pairs. These would take into account the idiosyncrasies of a particular hardware platform and could then accordingly optimize for things such as cache performance or memory performance. If these interpreter/transformer pairs were keyed by a specific platform identifier, then it should be possible for the SIND dispatcher to instantiate a more highly optimized interpreter on those platforms, rather than the more generic instruction set interpreter.

## 9.3 Advanced Security Assertions

Future applications of the SIND work related to security could include the protection of higher-level operating system services, protecting against heap overflows, function pointer overwriting, and other trespasses of process memory. Further applications could include work with proof carrying code and other formal security methods.

## References

[1] Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical report, Sun Microsystems Labs, June 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[3] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[4] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates. Fx!32 a profile-directed binary translator, 1998.

[5] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zang. Automatic detection and prevention of buffer-overflow attacks. *7th USENIX Security Symposium*, 1998.

[6] Mike Davidson. Lxrun. http://www.ugcs.caltech.edu/˜steven/lxrun/.

[7] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[8] Brian N. Handy, Rich Murphey, and Jim Mock. Linux binary compatibility. In *FreeBSD Handbook*.

[9] A. Klaiber. The technology behind Crusoe processors, 2000.

[10] Sun Microsystems. proc - /proc, the process file system. In *SunOS 5.8 Manual*, chapter 4.

[11] Sun Microsystems. ptrace - allows a parent process to control the execution of a child process. In *SunOS 5.8 Manual*, chapter 2.

[12] Sun Microsystems. The Java Hotspot performance engine architecture, 1999.

[13] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.