

Security in Dynamic Execution Environments

Hajime Inoue
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
hinoue@cs.unm.edu

September 27, 2001

Copyright © 2001 by Hajime Inoue
ALL RIGHTS RESERVED

Contents

1	Introduction	3
2	Trends in Architecture	3
2.1	The Road to VLIW	3
2.2	Implications of VLIW	5
3	Trends in Software	5
3.1	Just-in-Time Compilation	6
3.2	Dynamic Specialization	6
3.3	Instruction Level Parallelism	7
4	Profiling	7
4.1	Frequency and Time Profiling	7
4.2	Heap Profiling	8
4.3	Path Profiling	8
4.3.1	In Hardware	8
4.3.2	In Software	8
4.4	Modeling Program Behavior	9
5	Security	9
5.1	Java Security	10
5.1.1	Typed Assembly Language	10
5.1.2	Sandboxing	11
5.1.3	Sample Java Security Exploits	12
5.1.4	Anomaly Detection	12
6	Proposed Research	13
6.1	Preliminary Results	13
6.1.1	Dynamic Sandboxing	13
6.2	Characteristics to Examine	15
6.3	Implementation	15
6.4	Analysis of Program Behavior	16
6.5	Dynamic Optimization and Automated Response	17
6.6	Evaluation of Possible Results	17
6.7	Contributions	17
7	Related Research	18
7.1	Application Specific Intrusion Detection	18
7.2	OGI: Immunix and the Blinded project	18
7.3	UNM: Computer Immune Systems	19
8	Conclusion	19

1 Introduction

Trends in computer architecture and in language design and implementation are resulting in dynamic execution environments. A program's environment is the interface and implementation of the system hosting it, whether it is an operating system, interpreter, or virtual machine. A dynamic execution environment (DEE), a term I am introducing, is an environment in which the implementation or interface, through extension, is continually changing. This can include mutability of the hosted program as well, since changing a program's behavior changes its interactions with its environment.

The popularization of the Internet is increasing the exposure of applications to a potentially hostile environment. Programs are more vulnerable because systems are more accessible to attackers *and* because programs are more mutable, and inserted code is difficult to distinguish from the original executable. DEEs can be used to improve security, however, by taking advantage of information required for optimization. Intrusion detection and optimization both rely on the notion of "regular" program behavior – a program's behavior can be predicted and that information used to improve performance. Similarly, deviations from predicted behavior can indicate an abnormal program state. My goal is to show that this approach is effective by implementing it in a DEE, probably a Java virtual machine. My proposed research touches upon profiling, dynamic optimization, anomaly detection, and automated response. The result will be a prototype in which an optimized program is its own security profile.

Leveraging information used for optimization for security is a novel idea. Security is viewed by users as a performance impediment. Systems incorporate the minimum of security features to avoid annoying them. The question my research asks is, "Can optimizations used in DEEs be leveraged for anomaly detection?" With the construction of this system, I hope to show a system in which security is enhanced "for free", or at least with very little penalty.

To justify this view on DEEs and security, I explain below why DEEs are inevitable and how they provide opportunities for enhancing security. First I argue that DEEs are already common and will become ubiquitous with the adoption of future computer architectures and platforms. After that, I provide overview of security, with particular emphasis on Java security models and problems. A short description of profiling follows because both optimization and security require information on program behavior, which is generally obtained through profiling. The penultimate section is a description of the research program. I conclude with a short description of current work in application intrusion detection and a comparison of my work with related research by Calton Pu's Oregon Graduate Institute (OGI) group and those of others in my group, Stephanie Forrest's Computer Immune System group at the University of New Mexico (UNM).

2 Trends in Architecture

Architectures today use two mechanisms to exploit aspects of program behavior: cache and branch prediction. For example, examine the architectures of the Pentium 4 and Athlon [18, 32]. Both include multiple levels of cache and also use sophisticated logic to predict the direction of branches based on past behavior. Caches exploit the fact that most programs access the same locations in memory over and over. Likewise, branch prediction takes advantage of the fact that most programs follow the same execution paths again and again. These are just heuristics, yet these heuristics are now fundamental in processor design. Future processors will exploit dynamic behavior even more. They will gather more information about the code they are executing to do so.

2.1 The Road to VLIW

In the early 1980's David Ditzel, David Patterson, and others observed CPUs were devoting an ever increasing amount of resources to complicated instructions [55, 59]. Occasionally, this helped the language implementer – it was easier to map a language to an architecture with a large set of powerful, general instructions than to one with a smaller set of less general ones. This so called CISC (Complex Instruction Set Computing) approach also allowed code to be very compact, a large benefit considering the small memories of the time. Unfortunately, a complicated instruction set requires supplementary logic for accounting and microcode. For example, the VAX had over 200 instructions and 16 addressing modes. It is so complicated that the early iterations of the architecture were multi-chip units. The

consensus at the time was that architecture would move to a style called High-Level Computer Architecture [28]. These architectures had obvious mappings from high level languages to native code – in effect, their assembly language is a high level language. For example, the Burroughs B5500 was built so that COBOL and ALGOL mapped almost directly on to its instruction set [41]. Obviously, the consensus was wrong.¹

Reduced Instruction Set Computing (RISC), was an effort to rededicate resources to actual computation. The result was an architecture with a fixed instruction length, a smaller number of instructions, fewer addressing modes, and instruction caches in place of microcode.² Because chips were simpler, they were easier to design and optimize. Compilers were easier to write because, with fewer choices, mapping the language down to the instruction set and then optimizing the resulting code was straightforward. RISC quickly gained acceptance and became the dominant philosophy in architecture design. Patterson and Sequin, in the paper introducing their RISC-I processor, stated:

We have convinced ourselves that complicated addressing schemes are not a vital part of high-throughput machines... We have taken out most of the complexity of modern computers without sacrificing much in code density while improving performance [55].

The RISC philosophy carried the day. In recent years almost all commercial processors have been built around the RISC paradigm. The only major CISC architectures left are those built around the IA-32 instruction set. Even those include RISC elements; the Pentium III, Pentium 4, and Athlon all decode instructions into smaller “micro ops” before execution.

Modern CPUs, whether they are CISC or RISC, are pipelined and super-scalar. Examples of both styles of architecture today, such as the Compaq Alpha and the Intel Pentium III, are immensely more complex than their first exemplars. Ignoring die shrinkage, this complexity is forced because the speed of the clock can be increased only by decreasing the amount of work done per cycle. Pipelining does this by creating an instruction completion “assembly line”. Each instruction is executed as a sequence of several smaller steps. A typical chip pipeline of five stages is described in [28]: Instruction Fetch, Instruction Decode/Register Fetch, Execute/Address Calculation, Memory Access, and Write Back. Unrelated instructions are processed in parallel.³ Although each instruction in this example may take five clock cycles to process, an instruction is finished every cycle. Non-pipelined architectures, in contrast, may take several cycles for each instruction. Faster clocks have pushed this principle to the extreme. The Pentium 4 uses a twenty-stage pipeline, for instance.

The second way to increase performance is to exploit instruction level parallelism (ILP). Super-scalar architectures contain multiple functional units. Instructions, once decoded, are assigned to a free unit. This potentially allows the completion of several instructions per clock cycle. When memory was as fast as the CPU this approach worked well. Unfortunately, today’s processors are much faster than memory. This is commonly known as the “memory gap” [28]. Sophisticated logic is required to keep multiple pipelines fed and running. This logic is used to reorder instructions, predict branches, and allow for out of order execution and completion. It is the need to coordinate the activities of numerous components that makes modern processors complicated.

In the early 1990s the increasing complexity of processors caused a backlash – an architectural style named Very Long Instruction Word (VLIW) gained prominence. VLIW was introduced in the mid 1980s. However, its spread was hindered by static compilers that could not produce code approaching the calculated architecture speed.⁴ Ditzel, a major proponent of RISC, became the largest proponent of VLIW, this time using dynamic compilation.

VLIW is an attempt to get back to the RISC ideal of devoting resources to the computation instead of overhead. A VLIW instruction consists of a bundle of smaller instructions, each corresponding to a functional unit on the CPU. The processor no longer needs a complex instruction decoder to parcel out instructions to the various units. Similarly, no attempt is made to reorder instructions or deal with data dependencies. All of the coordination previously performed by the chip is now pushed to the compiler. Thus, more of the chip’s logic is devoted to computation.

¹I am ignoring some of the difficulties compiler writers had in mapping high level languages on to CISC architectures. Often, architectures were built with one language in mind, making compilers for other languages difficult to write. Going back to the B5500 example, Fortran ran very slow [41]. Also, the sheer complexity of the instruction sets made optimization difficult [70].

²Register windows are another important part of RISC, but are not discussed here.

³Instructions dependent on the results of other instructions cause pipelines to stall until the result is known.

⁴John Ellis’s dissertation on the VLIW Bulldog compiler describes the difficulties of compiling general purpose code for VLIW architectures [22].

2.2 Implications of VLIW

VLIW machines are designed to run fast by reducing hardware complexity. Therefore, the compiler handles instruction interdependencies instead of the processor. Removing this logic from the chip makes it easier to add execution units – there is no longer any overhead for accounting.

The price of hardware simplicity is software complexity. Static compilers for VLIW must be intelligent, and since they have to substitute compile-time software for runtime hardware, they are very complex and not responsive to runtime events. The compiler for a VLIW system requires more than just knowledge of the instruction set – it needs to know the exact properties of the functional units to schedule instructions efficiently. The optimal schedule of instructions for maximum throughput is dependent on the length and characteristics of the functional units. These characteristics differ over implementations of a single instruction set, but are hidden in other architectures.⁵ Because of this VLIW binaries run well or perhaps at all only on a specific implementation of an architecture. A binary compiled for one implementation may have completely different performance characteristics when run on a different implementation.

Of the several new architectures that have recently been announced, Transmeta’s Crusoe [37], Sun’s MAJC (Micro-processor Architecture for Java Computing)[48], Intel’s Pentium 4 [19], and Intel’s Itanium [2], the Crusoe and MAJC are VLIW. The Pentium 4 is a more sophisticated implementation of the IA-32 instruction set, while the Itanium is a hybrid between VLIW and traditional designs.

Both the Crusoe and MAJC solve the problem of processor timing incompatibilities by pushing compilation to runtime. The Crusoe and MAJC target the IA-32 and Java instruction sets, respectively. These are external instruction sets, however. Only one program is ever compiled to the native instruction set: the emulator. The emulator works similarly to many Java Virtual Machines (JVM) today. Clearly, this allows for maximum flexibility. Nothing, not even the instruction set, need remain constant during the lifetime of the architecture.

Intel is following a unique strategy with their processors. The latest version of the x86, the Pentium 4, uses dynamic translation to “micro-ops”, similar to the approach described above. The difference is that instructions are translated in hardware and stored temporarily in a cache. In effect, it has an emulator in hardware. If the cache is ever flushed, the effort of translation is wasted. Intel’s variation on VLIW, EPIC (Explicitly Parallel Instruction Computing), used in the Itanium, is meant to run executables natively. The Itanium does instruction scheduling in hardware – the compiler provides only dependency information. In short, the Itanium only goes halfway.

The inevitable conclusion is that for both performance and compatibility, architectures are moving to hybrid software/hardware implementations. DEEs will carry the day.

3 Trends in Software

The addition of dynamic aspects to computer architecture is dwarfed by the continual increase in software. Programming languages over time have grown in abstractness, expressiveness, and portability. The intended effect is what Richard Gabriel calls “compactness” – the ability to express an idea in a smaller number of lines than in previous, lower level languages [26].

These characteristics force implementations toward dynamic solutions. For example, in object oriented programs, binding of specific instances to specific implementations is done at runtime. Similarly, in programs written in functional programming languages, direct jumps to routines cannot be calculated until runtime since functions are often not created until runtime. Unfortunately, abstraction and dynamic behavior hinder performance. Most popular languages are still compiled because interpreted ones are perceived as slow. Indeed, even some of the more dynamic features of C++, like Real Time Type Inferencing (RTTI), are avoided because of the overhead required [66].

The increasingly dynamic aspects of both hardware and software place a burden on language runtime implementers. The unfortunate price of these recent trends is performance. Interpreted Java is notoriously slow. Java on a processor like Crusoe introduces two layers of emulation, that of the Java and IA-32 instruction sets.⁶ How is it possi-

⁵For example, the Intel 80386 is not pipelined while the Pentium 4 has a 20 stage pipeline.

⁶In principle only one layer of emulation is necessary. Transmeta could develop an emulator for Java that translates directly to the lower level VLIW architecture. At this time, however, they have yet to do so.

ble to achieve acceptable performance on a system like this? Just-in-time (JIT) compilation and dynamic specialization allow increased execution speed by taking advantage of “the specific”.

3.1 Just-in-Time Compilation

Because interpreted Java is unacceptably slow, VM implementers employ JIT compilation to improve performance. In a simple JIT compiler, a class’s methods are translated from Java byte code to the native instruction set when the class is loaded. Subsequent invocations to methods of that class execute the native code instead of interpreting the Java byte code. JIT compilers are therefore *lazy* compilers – they compile only the classes that are required for an application to run. In short, JIT compilers take a portable program and specialize it to a particular architecture.

Java Virtual Machines (JVMs) in use today are more complicated. Three interesting implementations are: IBM’s Jalapeno [12], Intel’s Open Runtime Platform (ORP)[17], and Sun’s HotSpot [47]. Jalapeno is written in Java. A program called a *boot-image writer* constructs the necessary services – class loader, object allocator, and JIT compiler – and saves the image as an executable. ORP is a research platform of implementing and testing JIT compilers for Java and Java-like languages such as Microsoft’s C#. Sun’s HotSpot is the industry reference. All include more than one compiler and profile Java programs as they run. They use profiling information such as the number of method invocations and presence of loops within methods to selectively recompile portions of the Java program with amore optimizing, but slower static but online compiler. A combination of compilers is used because the time spent compiling cannot be used to execute the Java program. It is wasteful to spend time optimizing code that is executed only rarely.

JIT compilers are used in hardware too. The two VLIW chips described above run all software in emulation. Both rely on JIT compilers to compile and cache code from their chosen instruction set to their native one. Crusoe uses profiling as well. One benchmarker noted that performance on some programs increased after many executions, almost reaching parity with the equivalent Intel processor [49].

Dynamo, another kind of VM, conclusively demonstrates that JIT compilers with profiling are useful [68]. Dynamo is a VM written by HP that uses profiling and a JIT compiler to translate programs written for their PA-RISC architecture to PA-RISC. Amazingly, programs executing in emulation in Dynamo run faster than programs running natively. With Java JIT compilers, a portable instruction set is compiled to a native one. Dynamo specializes code by performing optimizations using information only available during execution.

3.2 Dynamic Specialization

JIT compilers are not the only way to increase performance. Before Java popularized JIT technology, other forms of feedback-directed optimization (FDO) were used to improve performance [64]. These techniques alter a program based on its previous dynamic behavior. By studying the behavior of processes, programs can be recompiled using information derived from profiling to benefit the most frequent execution path. Altering the instruction scheduling can optimize a particular path through a program, possibly penalizing other paths. For example, procedures can be inlined based on their execution frequency.

Many commercial compilers exploit profiling data during recompilation. Morph, a research system, can profile and optimize binaries without source [71]. This is only beneficial, however, when the resulting executable maintains the same behavior exhibited by the program during its profiled run. In runs manifesting different behavior, feedback-directed optimization can *hinder* performance.

Massalin’s Synthesis kernel and the continuing Synthetix project at the OGI demonstrate that tailoring a program to its execution environment at runtime can produce dramatic improvements [58, 56]. Synthesis used three methods of generating specialized code [58]:

- Factoring Invariants to bypass redundant computations
- Collapsing Layers to eliminate unnecessary procedure calls and context switches
- Executable Data Structures to shorten data traversal time

The first two are applicable in many systems today. For example, in file system routines many variables become fixed after an *open* is invoked. By recompiling these invariants into new temporary routines, speed is increased

enormously [56]. Collapsing layers can be used in systems like network stacks. Although separation of layers is a useful design concept, the actual implementation is faster if the procedure calls and context switches separating them can be factored away. More recently, a group at Carnegie Mellon demonstrated that a high level specification of TCP/IP in ML could be compiled to native code with practical performance characteristics [9].

Recently, specialization has been applied to Java programs using similar strategies as Synthetix [61]. The authors report speedups of 1.44 to 5.13 times in an image filtering program.

Although not often used now, dynamic specialization will become more common in the future as performance concerns conflict with the increasingly abstract nature of software.

3.3 Instruction Level Parallelism

VLIW is designed to exploit instruction level parallelism (ILP). The dynamic transformation techniques discussed above are designed to create larger basic blocks – more specialization means less branches. Large blocks are necessary for efficient VLIW compilers. Unfortunately, the transformations involve tradeoffs. Performance increases come from optimizing the frequently executed path at expense of others. This cannot be successful without intimate knowledge of process behavior. Profiling adds additional overhead which current implementors are loathe to pay, but in order to benefit from architecture improvements, increased use of profiling is inevitable.

4 Profiling

Program profiling is used to gain information about the dynamic behavior of a process. Typically an executable is run in either a special environment or the code is modified with instrumentation to emit profiling data. Such data can reveal code “bottlenecks” – places in the code where improvement would yield significant performance gains. Also, many compilers now accept profiling data for use in recompiling code with optimizations [64].⁷ Instrumenting memory usage can reveal memory leaks. Purify and the Mozilla group’s Boehm Garbage Collector package are commonly used examples [27, 51].

Profiling is often thought of in this narrow sense. Yet modern processors use several tricks to leverage the dynamic behavior of processes. Caches and branch prediction are examples, as described in Section 2. New commercial and research platforms extend this to trace caches and path prediction for speculative multithreading [18, 60, 48]. In software, more attention is being paid to interprocedural and global behavior of programs because instruction level parallelism in VLIW cannot be exploited without optimization over multiple basic blocks. Again, path profiling allows greater optimization than previous techniques. The following is a survey of current profiling technology.

4.1 Frequency and Time Profiling

To show the capabilities of a typical modern profiler, I describe Paradyn-J, a profiler for Java applications in a modern VM [52]. Paradyn-J adds instrumentation code to Java code as well as collecting data from the VM itself. Paradyn-J collects data on the number of times a method has been invoked, the time spent in the method, the time spent waiting for IO in each method, the time spent in the VM while in the method, as well as statistics on the time spent compiling the byte-code to native code. The time spent in the VM corresponds to time spent in the kernel for normal applications. In general, profilers allow the counting of time specific events.

Several processors have added hardware support for profiling. UltraSPARC processors count events such as instructions executed, cycles, stalls of various types, and cache misses [3]. Some implementations of the Compaq Alpha also support profiling in hardware [5]. The ability to profile program behavior with little performance degradation allows much more analysis – profiling can be done on production systems rather than only on developers’ test runs.

⁷The Gnu Compiler Collection (GCC) is the notable exception.

4.2 Heap Profiling

Memory usage is another important aspect of modern profilers. Paradyne-J tracks object creation by method and object lifetime. Java applications are different from C/C++ applications because they use garbage collection. Frequently, VM garbage collectors include heap compaction as a secondary benefit. The garbage collector is instrumented to show when it is called and what it collects.

No commonly used processors implement fine grained memory profiling in hardware. However, caches and the virtual memory system can be viewed as a sort of coarse grained memory profiling system. The content of a processor's data and instruction caches reveals the current working set of the program, and the data being manipulated by the working set. Cache misses can also be recorded to gain information about the timing of working set changes and their sizes.

4.3 Path Profiling

Path profiling is different from the previously discussed types of profiling. It helps reveal global behavior and is therefore more complicated than frequency, time, or heap profiling. Rather than count events, path profiling reveals execution pathways – the route a process takes through the code as it executes. In software, trace scheduling is used to globally optimize code. Similar to control-flow diagramming, it shows the “flow” a process actually takes rather than all possible routes. Modern microprocessors leverage two techniques from path path profiling: branch prediction and trace caches.

4.3.1 In Hardware

Branch prediction is seldom characterized as a profiling technique. It does, however, monitor the dynamic behavior of a program to make predictions about the future for performance benefits. It was the first fine-grained hardware technique to predict program behavior based on execution paths.

Hennesey and Patterson describe in *Computer Architecture: A Quantitative Approach* why branch prediction is necessary for modern processors [28]. First, they state that branches will arrive n times faster on an n -issue processor than on a 1-issue one. Next they cite Amdahl's law: the impact of control stalls will be larger on machines with lower cycles per instruction. Branch prediction is necessary to remove these stalls.

Early branch prediction used one bit to predict which branch to take. The decision was easy – predict the same branch direction that was taken previously on the same branch. This is often ineffective, and many machines now use “correlating predictors.” These take into account the branches taken before the current branch point to make the prediction. This is very simple path profiling in hardware.

Trace caches are a more recent and sophisticated idea [60]. Trace caches are a form of instruction cache. Instead of taking instructions from memory as is, instructions are cached in the order they are executed. Trace caches are effective because of locality, like any cache, and traces tend to correctly predict branch behavior. With the introduction of the Pentium 4, trace caches have moved into production systems. The Pentium 4 uses a trace cache as an instruction cache for “micro ops” [18].

4.3.2 In Software

Trace scheduling by compilers is similar to trace caches in hardware [28]. Profiling information is used to predict “hot paths” through code. Many programs spend 90% of their execution in just a few number of traces. These are the “hot paths”. Traces consist of a series of sequentially executed basic blocks. The code is recompiled so that traces are sequential in memory as well as in execution. Branches out of the trace are moved elsewhere so that the selected trace has is a large contiguous block of instructions. This has the same effect as in the trace caches above. The instruction cache now behaves like a trace cache, improving locality and improving latency on correctly predicted branches.

The profile required for trace scheduling must be constructed by examining execution paths. This is usually done by edge profiling. If basic blocks are viewed as nodes on a graph, edges are branches. A trace is constructed by following the most commonly executed edge out of a basic block [15]. These traces, since they are now contiguous, can be optimized as one large block.

A group at Microsoft research has pointed out that this heuristic is not perfect – bad traces can be constructed [7]. They have discovered another technique that builds slightly longer paths. They have extended this to what they call Whole Program Paths (WPPs). They use these path profiles to build a complete model of program behavior in the form of a context free grammar (CFG) [38]. They point out that this technique can be used for bug detection as well as feedback directed optimization. For example, the path profile of a program with a Y2K bug was substantially different with post 2000 dates than with pre 2000 ones [8].

The Dynamo project, mentioned above, uses path profiling techniques to create “hot paths,” which they call fragments [68]. A benefit of Dynamo over static compiler trace scheduling is that the profile used for optimization is generated by the process to be optimized.

Other profiling techniques are possible. I performed some initial experiments in Java profiling with Helfman’s Dotplot [16] and was able to identify some trace like behavior. Dotplot compares a program’s profile trace against itself. This allows one to see repetitions of similar behavior. Other tools borrowed from computational biology, like Baker’s Dup [6], may also be useful. Dup is able to find repetitions of similar strings in a larger input string.

4.4 Modeling Program Behavior

Sections 4.3.1 and 4.3.2 review how modern systems use profiling in both hardware and software to improve performance. Often profilers’ output cannot be used directly by static compilers – developers are required to guide and interpret output and then restructure their source to benefit. The restructuring process often requires the developer to construct new algorithms and modify data structures. Even though this process is less automated than one might wish, the the information provided by the profiler is useful because the space in which most programs execute is tiny compared with space of possible behavior. That is, most programs rarely exhibit the wide range of possible behaviors of which they are capable. Optimization is possible because of this.

Better models of dynamic program would be extremely valuable. Promising approaches are likely to be based on modeling the data from path profiling. The CFGs from WPPs move toward this. It might be useful to try simplifying these models down to finite automata from context free grammars or perhaps borrow from more physical models such as statistical mechanics.

5 Security

Systems that combine profiling and dynamic optimization will soon be ubiquitous. VLIW processors force JIT compilation at the architecture level. The increasing use of software components and mobile code will force it at the application level. The extensive profiling required by JIT compilers will be used by more sophisticated specialization techniques. Every layer – the architecture, the OS, and application – any interface providing services will be monitored and specialized. This provides an opportunity which is the focus of my dissertation research.

My research is primarily about security – not dynamic optimization. I plan to develop systems that leverage optimization for security, and so some background information on computer security, and Java security in particular, may be useful to the reader.

Traditional security concerns, as stated by Anderson in an early computer security paper, can be divided in to three categories [4]:

- **Unauthorized information release:** Information should not be available to unauthorized entities.
- **Unauthorized information modification:** The modification or deletion of data on a system should not be possible by unauthorized entities.
- **Unauthorized denial of service (DoS):** Entities should not prevent authorized users from accessing a system.

This view of security encompasses mechanisms – it assumes that policy and its implications are easily understood. Correct implementation of these mechanisms should result in a secure system. Unfortunately, even ignoring the fact that almost all large programs have bugs, program specifications are often insufficient or wrong. Programs may also have undesirable features such as backdoors. Even seemingly innocuous features, like debugging code, can have

disasterous consequences, as the Morris worm showed [21]. Secure systems should view both mechanism and policy as potentially faulty. The realm of computer security applies to both.

5.1 Java Security

Security in Java is usually framed as a discussion of how Applets' sandboxes prevent access to a client's hard drive. This is the most trivial of cases. Applets are Java's most visible application, but not the most important. Java is increasingly used on web servers [45, 46], or as middleware [44]. Airlines, brokerages, and health care providers rely on Java applications to manipulate sensitive data. Secure Java applications are clearly important to these organizations.

Java Security is a broad topic. A complete discussion would cover network as well as application security. Here I describe only the fundamental security features of the Java platform. Briefly, I mention how one implements policies using these basic mechanisms.

5.1.1 Typed Assembly Language

All of Java's security mechanisms rely on its verifiable byte-code. The Java instruction set consists of a form of typed assembly language (TAL), called byte code. Typing at the instruction level allows the VM to infer the type of every memory location and enforce strong typing at the instruction level. Others have noted that typed assembly language is a form of proof carrying code [50]. This type of proof leads to a straightforward verification mechanism.

A class is verified in the following way. When the class is loaded, the verifier performs four passes:

1. Checks that the class is a valid class file and if digitally signed, that the signature is genuine.
2. Checks that the symbol table is consistent. Consistency means that all the symbols have known types and that symbols referred to by other symbols are of the correct type.
3. Checks the code for correct typing and stack manipulation properties. This amounts to replicating the static type checking that is supposed to occur in the compiler, and ensuring that the code doesn't violate the stated bounds of the operand stack.
4. Checks typing for referenced classes. This pass is performed when a method is executed, rather than when it is loaded. Otherwise, each referenced class would need to be loaded as well.

The most interesting parts of verification occur in passes 3 and 4. Type checking is very important because it ensures that invalid pointer manipulation cannot occur. Verification also models stack behavior to rule out the possibility of undefined behavior through wrong typing or incorrect manipulation of the operand stack.

Some checks, such as dereferencing null pointers, array bounds checking, and type casting must be performed at runtime. The complete verification mechanism is somewhat more complex. Details of the process can be found in [40, 54]. With verification and runtime checks, however, the VM guarantees the following properties:

- Permissions on methods and variables are respected.
- Typing is satisfied.
- Programs cannot access arbitrary memory locations.
- There is no undefined behavior.

If the VM and verifier are implemented correctly, the class should obey all constraints of the Java language. Failure to follow the constraints results in throwing of exceptions which, unlike in many languages, must be caught. These mechanisms provide "safety" – code executes only through known and specified interfaces. Security is implemented through control of the interfaces, with the class loader and verifier being the most important.

5.1.2 Sandboxing

The concept of sandboxing was introduced by Wahbe et al. in the application of software fault isolation [69]. Sandboxing is a technique in which a system shields some of its resources from a program. Early Java releases had difficulty to configure sandboxes. It is now possible, however, to implement flexible sandboxing in Java. It is an amalgamation of six parts. The first two, the class loader and verifier, are mentioned above. The others are:

- The Access Controller manages access to the OS and other privileged resources in Java 1.2 and above.
- The Security Manager has responsibility for all resources on a system.⁸
- The security package is used to authenticate signed classes and is too complicated for detailed description here. See *Java Security* for documentation of the complete API [54].
- The key database stores keys used by the Security Manager and Access Controller.

The Security Manager controls the extent of the sandbox. By default, applications have no Security Manager. Applets have a very strict one. The Applet sandbox is the one commonly referred to when Java sandboxing is mentioned.

The Applet sandbox disallows all access to the host filesystem. It allows opening of sockets only to the host that supplied the applet. Clearly, this is too restrictive for many purposes. Many applets would be more useful if they allowed one to save data on a disk or communicate with more than just the original server. Changing the Security Manager provided to the Applet by the Applet runner changes the sandbox. Prior to Java 1.2, this was very difficult.

Inclusion of both the Access Controller and Security Manager in the Java libraries is somewhat redundant since they play the same role. The Security Manager is ill suited for expansion. It contains methods hardcoded for specific permissions, like `checkWrite` (for filesystems), `checkAccept` (for sockets), and `checkExec` (for processes). In Java 1.2, the Access Controller was introduced to generalize it – permissions are checked using Strings as arguments. This allows for the addition of privileges without modifying the class.

A sandbox is created in the Access Controller by providing it an instance of the Policy class. Policy is essentially just a collection of Permissions. A Permission is a name and a list of allowed actions. For example, a file permission consists of a pathname and a list of actions like “read” and “write”. Normal implementation of sandboxing results in the very familiar access matrix model of security described in many OS texts [67, 63].⁹ More complicated sandboxes can be created using Protection Domains. These result in multiple sandboxes where the sandbox differs depending on where the code resides.

Security in Java depends on correct implementation of the safety mechanisms as well as an appropriate policy. This is by no means an easy thing. Many of the security violation examples described below in Section 5.1.3 exploit verification bugs. Those will become rarer over time. The algorithm is well known and isn’t likely to change. The addition of dynamic compilation and optimization, however, complicates the situation. Even if the verifier is trusted, the compiler might not be. Dynamic optimizing compilers are much more complicated than byte-code verifiers. An attacker might be able to exploit buggy JIT compilers to execute arbitrary native code and bypass security mechanisms entirely. Adding correct permission checks is another problem. Whenever a new resource is added to a system, access to it must include the appropriate hooks for the Access Controller. Without them, there is no protection.

Furthermore, even if the mechanisms are correctly implemented, there is no protection without correct policy. Java security can be very complex. The usual simplifying choice is “all or nothing.” Either use the default application security model (no security) or the applet one (too restrictive). The default application model is a simple choice since any nontrivial program usually demands access to system resources. The alternative is to implement new resources, permissions, policies, and perhaps new protection domains. Doing so was impossible prior to Java 1.2 and is not easy now.

⁸This is a wrapper for the access controller in Java 1.2.

⁹The access matrix model is a matrix with resources on one axis and resource users on the other. Cells contain permissions. For example, the unix file permission schemes can be viewed as a matrix with a list of files on the horizontal axis and three categories, user, group, and other on the vertical axis. Each cell could contain read, write, or execute.

The Java security framework ignores denial of service attacks. It might be possible to stop these attacks using the sandboxing model (by using the Access Controller to monitor the system and modify policy dynamically), but no one has suggested this.

5.1.3 Sample Java Security Exploits

There are many exploits against JVMs. The ones reported here are all applet exploits and were reported after April 1997. There were several discovered prior to 1997, but I felt exploits against newly released software weren't as informative as attacks against mature software. Many of these exploits are against the verification mechanism:

- **October 1999:** A bug in the verifier in Microsoft's VM allows casting of values to unrelated types. This can be used to breach the sandbox and gain complete access to the client's computer.
- **April 1999:** A bug in all popular JVMs allows downloaded code to execute without prior verification. Malicious bytecode can subvert the type safety mechanisms to gain access to the client.
- **July 1998:** A Classloader bug allows an applet to redefine vital classes and lead to uncaught violations of the type system, potentially allowing breaches of the sandbox in Netscape 4.0.x.
- **April 1997:** An Applet can change its own signee to one that is trusted, allowing subversion of the sandbox in JDK 1.0

Others exploit the Security Manager:

- **October 2000:** URLConnection and URLInputStream do not implement Security Manager hooks. Applets can then access files using a string argument of the form file://filename to gain access to any file on the client, including directory listings on Netscape 4.5.x. The demonstration exploit implemented an HTTP server named Brown Orifice which gives an attacker unlimited access to a victim's files.
- **July 1998:** A bug in the Security Manager allows applets to turn off the verifier in Netscape 4.0.x.
- **August 1999:** A race condition in the Security Manager library of Microsoft's VM allows violation of the security rules. Applets can gain total access to the machine using this bug.

All the above exploits except for Brown Orifice were reported by the Princeton Secure Internet Programming team [23, 42]. Brown Orifice was discovered by Dan Brumleve [10]. CERT has issued three advisories pertaining to Java exploits. Other individuals post exploits to their web pages [1]. Some of these show DoS exploits, which do not breach the sandbox, and are annoying (removing them may require shutting down the browser), rather than dangerous.

Currently, almost all exploits are applet exploits. Apparently application exploits have not been considered a priority for crackers, possibly because the default application sandbox has no restrictions. There is only one known application exploit, a virus named Strange Brew [35]. It is interesting that attacks against specific applications are not reported, however. Large companies employing Java on the server may not publicize attacks. It is also possible that crackers still target easier prey – applications vulnerable to buffer overflow attacks. Java's verification checks on stack size and runtime checks on array bounds make this kind of attack impossible.

5.1.4 Anomaly Detection

The security mechanisms in Java cover Anderson's understanding of security. If one has confidence in the verification, security manager, and policy, one should have confidence in the security. This is unrealistic, however. From the list of exploits, it is clear that even in the relatively simple browser VMs there were many security holes. VMs are becoming more complicated. Further, Java applications on the server end are very large and used in critical settings. Security, especially for relatively untrusted code, comes from multiple redundant systems. Systems should have traditional security mechanisms that try to stop security violations based on policy. They should also have systems that recognize

known attacks and try prevent them. These are similar to virus protection systems that recognize signatures. The missing system is one which detects novel attacks which may pass through the first line of defense.¹⁰

Anomaly detection is a useful third system since it has the possibility of being lightweight until an anomaly is seen. Anomaly detection is often used in computer security. The usual idea is to find some statistical basis of normal and then measure the process or network (for network intrusion detection) for deviations from it.

Anomaly detection could address other concerns. Java's security model does little to assure that applications don't have back doors or other exploitable features that are byte-code verifiable but unwanted. Programs with these bugs might execute within the configured sandbox policy but also allow security violations. Anomaly detection may allow us to catch this behavior without performance burden.

6 Proposed Research

DEEs are arriving and their greater popularity is inevitable, but they are not ascendant yet. Their properties makes one ask the question, "Can the profiling required for dynamic optimization be used for anomaly detection?" The rapid popularization of Java should allow me to answer this question. I intend to use Java to investigate the uses of optimization for security purposes.

6.1 Preliminary Results

Several months ago, I modified a JVM under the GPL, Kaffe, to profile Java Programs. Runs of small programs revealed that the applications call very few unique functions in comparison to the total number of function invocations. This is also true if I require functions to have identical arguments in order to be treated as the same function call. Specifically, the graph of function vs invocation frequency is a straight line when plotted on a log-log scale – that is, it is a power law. See Figure 1 for an example using the canonical "Hello World" program. The number of novel function invocations for most programs decreases drastically over time, until the shutdown process invokes many novel functions.

These facts can be leveraged with respect to security. I altered Kaffe to accept the profiling information provided by the earlier modifications. Profiling information generated by earlier runs forms the *normal* profile for later executions. The modified Kaffe only allows methods that are listed in the profile to be invoked; the contents of arguments are ignored. This forms what I call a *dynamic sandbox*. This contrasts with the two standard Java sandboxes, described earlier in Section 5.1.2, which are inflexible.

The first test of dynamic sandboxing was to check that normal executions didn't provoke false positives. This was especially important since a detected anomaly caused Kaffe to abort. To test this, I generated profiles for an applet provided as a demo for Sun's development kit and a simple HTTP server that I wrote for this experiment. When dynamic sandboxing was enabled during normal execution, no false positives were detected.

The HTTP server included a simple backdoor which allowed an attacker to start arbitrary processes. The normal profile was generated without using the backdoor. With dynamic sandboxing enabled, use of the backdoor was successfully detected and trapped. A more complete description of these experiments is in [34].

Aborting the JVM on an anomaly detection is probably a bad response. False positives are to be expected in more realistic tests. A more sophisticated response hasn't been developed yet.

6.1.1 Dynamic Sandboxing

Unfortunately, these modifications to Kaffe resulted in substantial slowdowns. I used the interpreter instead of the JIT compiler because it was much easier to modify. I also kept the profile separate from the cache of loaded classes. This slowdown was not necessary. Dynamic sandboxing might result in *better performance*. This is the surprising result

¹⁰This is similar to immune systems found in vertebrates. The first line of defense are barriers: skin and mucus membranes in vertebrates, permissions and sandboxing in the computer realm. Notice that vertebrate second systems of memory B cells perform a similar role to virus detectors' signature scanners. Finally the innate immune system in vertebrates, the final piece of the puzzle, performs a similar role to my proposed third system for novel attacks. The automated response component is like the initial directed response of the adaptive immune system.

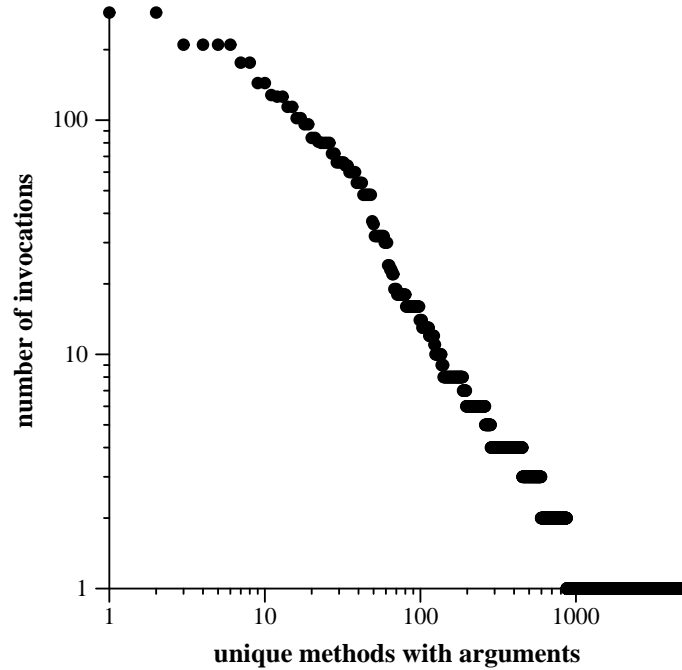


Table 1: Power law behavior of “Hello World” in Java

of using a profile that reflects the predicted behavior of the program. An optimized dynamic sandbox would use the profile to prefetch and compile classes intelligently before they are needed. Since after a short period the JIT method cache is the profile, there is no overhead. Consequently, this application of dynamic sandboxing might result in both better security and improved performance.

I intended to demonstrate this using Intel’s Open Runtime platform (ORP). Unfortunately, I have been unable to cache code within the profile itself due to absolute addresses in the native code. However, I have recently modified Intel’s Open Runtime Platform (ORP) to include support for dynamic sandboxing without the native code caching. Support for profiles was added by forcing the JIT compiler to emit the method class and signature before the method is compiled. This has low overhead since the JIT compiler compiles the vast majority of methods only once.¹¹ Adding dynamic sandboxing also involved modifying the JIT compiler. Before the JIT compiler invokes the compilation routine, it must now check to see if the method signature exists in the profile. Unlike in Kaffe modifications, there was no attempt to record or check method arguments. Also, this modification didn’t implement the optimization described above in which methods’ native code is cached in the profile. A full description of the implementation and its effectiveness and performance can be found in a paper submitted to the Recent Advances in Intrusion Detection conference [33].

I tested the modified ORP against the trojan HTTP server mentioned earlier and the Java Strange Brew virus [35]. Dynamic sandboxing found no false positives for the trojan HTTP server under normal operation. It did catch both the use of the backdoor and the virus. This implementation of dynamic sandboxing was also efficient – the slowdown on profile generation and sandboxing on the Olden benchmark suite was less than 2%, as shown in Table 2 [14, 13].

¹¹Methods may be compiled more than once depending on the recompilation policy. There is no recompilation policy in place for my modified version.

ORP Parameters	Mean user + system time in seconds (Std. Dev.)
Default (no sandboxing)	304.59 (0.26)
Logging the sandbox profile (-profile)	308.93 (0.35)
Dynamic sandboxing enabled (-sandbox)	308.03 (0.19)

Table 2: Efficiency of sandbox generation and protection on the Olden benchmark.

6.2 Characteristics to Examine

Although these preliminary results demonstrate that dynamic sandboxing can be effective against many attacks, the initial system cannot detect attacks when an application’s behavior is varied. Specifically, if the application’s profile uses a large proportion of the methods available to it, the profile will span much of the possible anomaly space, making detection of security violations unlikely.

Function call invocations are the first of many characteristics that are likely points of optimization in future DEEs. These characteristics must be retrievable from the process in an online and computationally efficient manner. Some promising ones are listed below:

- **Basic blocks:** Using the basic block as the profile element increases the possible anomaly space enormously. Furthermore, many optimizations work on basic blocks, making them targets of instrumentation in adaptive VMs.
- **Method arguments:** Instrumenting method arguments would result in a large method-argument space allowing finer distinctions between self and non-self. I have partially implemented this in Kaffe.
- **Method sequences:** System call sequences have been shown to be characteristic of specific processes [24]. Method sequences may show similar behavior. There are two obvious options. One can look at the sequence of calls as a list, logged in the order methods are entered. Alternatively, one can look at the sequence of calls on the stack on method entry. Either one may yield interesting information when analyzed with an n-gram like profiling scheme like the one used in the system call work. The ties to optimization here aren’t strong, though it is plausible that n-grams could be useful for locality and scheduling optimizations.

Another possible profile could be a finite state machine based on call sequence. A similar idea was applied to system call sequences by Michael and Ghosh [43].

- **Memory behavior:** Some applications have distinctive heap allocation and garbage collection behavior. Garbage collection is increasingly prevalent and integral in many performance sensitive applications. Happily, there is much work being done on adaptive GC methods. Most GCs have hooks for instrumentation as well. Instrumenting memory management might reveal anomalies not seen with other techniques.
- **Whole program paths:** Larus et al showed that programs spend 90% of their time in a small number of “hot paths” [38]. Paths reveal a large amount of information about the global behavior of an application. Also, trace caches in hardware and Dynamo-like systems already keep track of this behavior [68].

This is not an exhaustive list. It is likely I will find other useful profiling characteristics. Any optimization is of potential interest.

6.3 Implementation

ORP will be the base of my implementation. It has a clean design and is meant to be a research platform. Unfortunately, ORP cannot run the vast majority of popular Java software because it uses the GNU cleanroom implementation of the standard Java libraries called CLASSPATH [25]. CLASSPATH is incomplete, especially in its AWT (graphics library)



Table 3: Screenshot of Helfman’s Dotplot program showing behavior regularities in the Java “Hello World” program. Dark areas correspond to repeated behavior.

implementation. CLASSPATH is advancing quickly though. GCC 3.0 incorporates a static Java compiler which uses CLASSPATH as its library. Redhat apparently has several engineers devoted to its development.

Until CLASSPATH’s completion, I am resigned to a parallel path implementation strategy. I will use ORP to identify optimizations and design profiling strategies. For experiments on real applications, however, I will need to instrument bytecode. Using a package such as BIT or BLOAT [39, 53], Sun’s standard libraries and JVM can be used, allowing aquisition of data from any Java application.

Also, I am not wholly committed to Java. Dynamo-like systems will continue to gain popularity.¹² If one is found that can be suitably instrumented, I may employ it.

6.4 Analysis of Program Behavior

The extensions to the VM and data gathered through code instrumentation will allow the analysis of dynamic behavior of programs. Knowing how programs behave is essential for determining what optimizations will be useful, and hence, what information will be available for anomaly detection.

For my first investigations I used Jon Helfman’s dotplot program [16]. I was able to detect large repeatedly executed sections of code as can be seen in Table 3. One can see, even in the canonical “Hello World” expected to have little repeated behavior, that there are large sections of repeated behavior on several scales. Dotplot is a coarse tool though.

I speculate that the best tools will be the optimizations themselves. As I stated in the introduction, optimizations are possible because they make predictions about the future behavior of the application. Analyzing when and how the optimizations are useful may allow me to develop better models of both normal and abnormal program execution.

Any model of program behavior must be informed by empirical data – that is, the execution of real programs. A variety of “real world” Java applications must be monitored, during both normal and abnormal behavior. Unfortu-

¹²For example, a group at UNM is currently implementing a SPARC version of Dynamo.

nately, there are not a lot of widely spread Java applications. Most Java applications are written and used internally by corporations as middleware. This is another motivation to move to a Dynamo-like system.

An easily conducted experiment is to replace the University of New Mexico Computer Science Department's FTP server with my own instrumented Java version. Also, I will contact other organizations about obtaining data on the dynamic behavior of their internal Java applications. This data should also help in devising responses to anomalies.

6.5 Dynamic Optimization and Automated Response

Once I have identified useful optimizations, I will either borrow or reimplement them in the modified VM. The modified VM will at this point detect anomalies under replicated experiments. The focus will now be on effective responses.

Automated response is necessary for anomaly intrusion detection for two reasons. First, because false positives are inevitable, the system should have other means to determine between the two and continue execution after false positive anomalies. Second, systems should ideally defeat, rather than just detect, security violations.

Most intrusion detection systems don't implement an automated response – they rely on operator intervention. IBM has a system for virus detection and removal [36]. Like most virus detection systems, it concentrates on scanning static binaries, rather than stopping the infected processes. Mark Burgess's *cfengine*, a system administration tool rather than intrusion detection tool, can respond to some anomalies [11]. For example, if system's disk becomes full, *cfengine* can run a script which removes nonessential files. Of course, these anomalies are more fixed and the responses more inflexible than in the ideal intrusion detection system.

One possible response is to focus on IO methods. These are usually native and are always able to throw exceptions. It may be possible to implement response simply by translating security violations into IO errors. A flexible way to do this is to have anomalies throw Exceptions of the same type stated in the Java libraries, but append a set of security interfaces so that fine-grained response by the application is possible, but not required.

Another option is to use dynamic rewriting to insert hooks into code accessing a custom Access Controller. The Controller would monitor the profile and dynamically control the new permissions through the introduction of a meta-policy – the programmed response to anomalies.

A third idea, borrowed from Somayaji's pH project, is to insert exponentially increasing delays after each anomaly, effectively "freezing" attacks [65]. A description of pH is described below in Section 7.3.

6.6 Evaluation of Possible Results

A logical end for this research is creation of a prototype VM that effectively leverages optimization for security. The prototype would leverage several optimizations with them covering the several different areas of program behavior listed above (GC behavior, arguments, etc).

This is assuming a positive result. Unfortunately, there is the distinct possibility that optimization cannot be leveraged for security in general. In that case, I should show in detail why the optimizations considered do not lend themselves to anomaly detection. Also, I should show in general why optimizations cannot be useful in anomaly detection.

Another possibility is that some optimizations will be useful and some will not. A description of why each optimization succeeds or fails for security will be required, along with a generalized explanation why some optimizations are useful and others are not.

Regardless of the outcome, I should be able to predict from a described optimization and my developed model of program behavior how well it will perform when leveraged for anomaly detection.

6.7 Contributions

The contributions of my proposed research are in the areas of program profiling, anomaly detection, computer security, and possibly dynamic optimization. Assuming a positive conclusion, the principle contribution will be demonstrating that dynamic optimization techniques can be leveraged for security. Further, I will show implementing that anomaly detection security at the application level is feasible and effective. With any result, advances will be made in modeling program behavior and global program profiling. Results may also suggest where new profiling features in hardware

would be effective. Finally, I may discover novel optimizations by showing that they are useful for anomaly detection, although this is not a main goal.

7 Related Research

The proposed research spans several subdisciplines of computer science. The optimistic result of the research should be an instance of application specific intrusion detection. There is a small amount of research in the area, mostly dealing with custom built intrusion detectors for prespecified applications. Two groups conduct research similar in many respects to my own. Calton Pu's OGI group worked on a project called Immunix with similar goals and superficially similar mechanisms. My own group, Stephanie Forrest's Computer Immune System group, has worked on two lightweight anomaly intrusion detection systems that work on similar principles.

7.1 Application Specific Intrusion Detection

Application intrusion detection systems (AppIDS) are an offshoot from operating system intrusion detection systems (OS IDS). Operating systems intrusion systems are now quite common, with the first system described by Dorothy Denning in 1987 [20]. More recently, there has been interest in intrusion detection at the application layer, with the idea that the semantics of the application may be used to detect more attacks. Robert Sielken's masters thesis at the University of Virginia addressed this issue [62]. Unfortunately, the AppIDS described by Sielken are mostly rule based and their construction is not at all automated. Although the goals of Sielken's research are similar to my own, the approach is very different. The security systems described by Pu's and Forrest's group, although aimed at lower levels of the execution platform, are more similar to my own.

7.2 OGI: Immunix and the Blinded project

In December 1996 Calton Pu presented the paper "A Specialization Toolkit to Increase the Diversity in Operating Systems" at the International Conference on Multiagent Systems (ICMAS) during a workshop titled "Immunity Based Systems" [57] This was the seminal paper of the Blinded project. The ideas presented during the talk have strongly influenced my proposal. I present a summary of the Immunix group's research and progress followed by a comparison with my own course of work.

The Blinded project has its roots in the earlier Synthetix project. The Immunix group proposed extending the Immunix toolkit created for the Synthetix project into the security realm. The thrust of their argument is a thoroughly biological one: diversity is good. There are two reasons for this. First, it is difficult to attack systems when the code and variable layout are unknown. Second, it is more difficult to attack successfully every system when each is different.

They aimed for two sorts of attacks, which they termed "hard-wired" and "contextual". Hard-wired attacks take advantage of the machine representation of a program. An example of this is a buffer overflow. Contextual attacks execute programmed functionality outside the original intentions of the implementators. They give the example of the Morris internet worm [21], which exploited a debugging feature in sendmail.

For hard-wired attacks, Immunix would vary the layout of code and variables. For contextual ones, the specializer would rely on heuristics. Specialized code is code and variables that are commonly seen (executed). A contextual attack would execute code or use variables that are not seen often and are therefore not specialized. The system would notice this and deal with it appropriately.

They speculate that the system would dynamically choose a level of security. High risk features could be turned off (by disallowing specialization) during "war time," but left in "peace time." Determining what features and times are risky is not addressed.

They summarize the project as:

Our goal is to combine specialization with system monitoring research and provide dynamic adaptation to resist detected attacks. By introducing randomization into dynamic adaptation, we hope to increase operating system survivability by increasing the resistance to security faults as they occur [57].

Their work, however, does not appear to follow the path laid out in the original paper. Further research, driven by Crispin Cowan, concentrates on stopping buffer overflows with the “StackGuard” extension to GCC. StackGuard protects against these attacks by adding random values to stack frames whose values are compared before a program jumps to a return address. If the values are overrun, as they might be during an attack, the fault is trapped. Only one paper deals with dynamic adaptation for security purposes [29], and that, although theoretically generalizable, deals with StackGuard security faults.

My work is superficially similar. We both want to specialize code for security purposes. The difference is that they concentrate on diversity. Hard-wired attacks are more difficult with specialized code since most of these attacks rely on knowing the layout of the process in memory. Against a particular type of hard-wired attack, buffer overflows, they are very successful. The part of Immunix designed to handle contextual attacks, which is the part most similar to my own work, was never implemented.

7.3 UNM: Computer Immune Systems

Two systems from UNM’s Computer Immune System group reflect on my own research: pH and Lisys. Both are anomaly intrusion detection systems. Lisys is aimed at network intrusion detection, pH is aimed at the operating system level. I am hopeful that my system, aimed at the application layer, will complement the other two. Both also work, superficially, by building a normal profile by monitoring what is assumed to be typical behavior for some period of time. The profile is then used in comparison with online behavior, and anomalies are flagged when the two are different. A short summary is provided for both. For more detail, see [65, 31, 30].

Lisys is directly modeled on the immune system. To explain a bit simplistically, some number of random packet headers are generated by the system on each node of the network as candidate detectors. During the “tolerization period” each packet’s header observed on the network is compared with each randomly generated one. If there is a match, the matched randomly generated header is discarded and a new one is created in its place. This is the equivalent of the negative selection T cells undergo during development in the thymus. After the tolerization period, packets that match the generated detectors are flagged as anomalies. A signal is sent to the operator who can ignore it as a false positive which automatically kills off the random packet. If the operator confirms the anomaly, the random header becomes a memory header. This system supplies the second and third systems described above: it catches novel attacks and creates signatures which memorizes attacks it has seen before.

pH (for process Homeostasis) uses sequences of system calls to build its own profile of normal behavior. A profile of typical behavior is built by a sliding window of past system calls. The sequence of calls within the window is deemed normal. After the normal profile is built, sequences that don’t appear in the normal profile are flagged as anomalous. pH tries to deal with anomalies with automated response – it typically delays completion of anomalous system calls. Within some time window every anomaly increases this delay exponentially. pH is able to stop several attacks including backdoors and buffer overflows. It is a pure third system.

I envision my prototype will be similar to these two in that I am working toward an anomaly intrusion detection with normals built from running processes. It is different in that I am looking at the application level with a greater potential wealth of information. These two systems don’t make the bet that in this flood of information regularities used for optimization will also reveal regularities relevant to security.

8 Conclusion

DEEs will become ubiquitous. Dynamic optimization and profiling will necessarily grow as well. Along with this growth will be the need for secure computing. A positive conclusion to the proposed research will show the relationship between dynamic optimization and program behavior – and how to exploit it. It will also show that security can be had without the loss of features and performance – security will be an entrenched, intrinsic part of systems.

References

- [1] Hostile applet home page. <http://www.cigital.com/hostile-applets/index.html>.
- [2] The ia-64 architecture software developer's manual vol. 3 rev. 1.1: Itanium (tm); instruction set reference. <http://developer.intel.com/design/ia-64/downloads/24531902s.pdf>, 2000.
- [3] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.
- [4] J. Anderson. Information security in a multi-user computer environment. *Advanced in Computers*, 12, 1973.
- [5] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone, 1997.
- [6] Brenda Baker. A program for identifying duplicated code. In *Interface Foundation of North America INTERFACE '92*, March, 1992.
- [7] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
- [8] Thomas Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *Computer*, 33:57–66, 2000.
- [9] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack: A systems application of Standard ML.
- [10] Dan Brumleve. Brown orifice. <http://www.brumleve.com/BrownOrifice/>.
- [11] Mark Burgess. Cfengine. <http://www.iu.hioslo.no/mark/cfengine/>.
- [12] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serano, Harini Srinivasan, and John Whaley. The jalapeno dyanmic optimizing compiler. In *JavaGrande*, 1999.
- [13] B. Cahoon and K. S. McKinley. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX, October 1999.
- [14] M. C. Carlisle and A. Rogers. Software caching and computation migration. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbra, CA, July 1995.
- [15] Pohua P. Change. Trace selection for compiling large c application programs to microcode. In *21th Annual Workshop on Microprogramming and Microarchitecture (Micro 21)*, pages 21–29, November 1988.
- [16] Ken Church and John Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993.
- [17] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimization. In *PLDI*, 2000.
- [18] Intel Corporation. Intel pentium 4 optimization manual. <http://developer.intel.com>, 1999.
- [19] Intel Corporation. Ia-32 intel architecture software developer's manual volume 1: Basic architecture. <http://developer.intel.com/design/Pentium4/manuals/24547002.pdf>, 2000.
- [20] Dorthy Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222, 1987.

- [21] Mark W. Eichin and Jon A. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, Oakland, Ohio, 1989.
- [22] John Ellis. *Bulldog: A Compiler for VLIW Architecture*. MIT Press, Cambridge, MA, 1986.
- [23] Edward Felten and Andrew Appel. Princeton secure internet programming. <http://www.cs.princeton.edu/sip/history>.
- [24] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.
- [25] Free Software Foundation. Classpath. <http://www.classpath.org>.
- [26] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.
- [27] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.
- [28] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2 edition, 1996.
- [29] H. Hinton, C. Cowan, L. Delcambre, and S. Bowers. Sam: Security adaptation manager. In *Proceedings of the Annual Computer Security Applications Conference*, Phoenix, AZ, December, 1999.
- [30] Steven Hofmeyer. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.
- [31] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
- [32] Advanced Micro Devices Inc. Amd athlon processor architecture. <http://www.amd.com/products/cpg>, 2000.
- [33] Hajime Inoue and Stephanie Forrest. Anomaly instruction detection at the application layer. Submitted to Recent Advances in Intrusion Detection, 2001.
- [34] Hajime Inoue and Anil Somiyaji. Security just in time. Submitted to Hot Topics in Operating Systems 8, 2001.
- [35] Landing Camel Intl. Codebreakers. <http://www.codebreakers.org>, 1998.
- [36] J. Kephart, G. Sorkin, M. Swimmer, and S. White. Blueprint for a computer immune system. In *Proceedings of the Virus Bulletin International Conference*, San Francisco, CA, 1997.
- [37] A. Klaiber. The technology behind crusoe processors. <http://www.transmeta.com/crusoe/download/pdf/cru-soetechwp.pdf>, 2000.
- [38] James R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.
- [39] H. Lee. Bit: Bytecode instrumenting tool, 1997.
- [40] Tim Linholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, 1999.
- [41] Alastair J.W. Mayer. The architecture of the burroughs b5000 - 20 years later and still ahead of the times? *ACM Computer Architecture News*, 1982.
- [42] Gary McGraw and Ed Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, Inc, 1999.

- [43] Christoph Michael and Anup K. Ghosh. Using finite automata to mine execution data for intrusion detection: A preliminary report. In *Recent Advances in Intrusion Detection*, pages 66–79, 2000.
- [44] Sun Microsystems. Java as middleware. <http://industry.java.sun.com/casestudies/>.
- [45] Sun Microsystems. Java server pages success stories. <http://java.sun.com/products/jsp/success.html>.
- [46] Sun Microsystems. Servlet success stories. <http://java.sun.com/products/servlet/success.html>.
- [47] Sun Microsystems. The Java HotSpot performance engine architecture. <http://www.java.sun.com/products/hotspot/whitepaper.html>, 1999.
- [48] Sun Microsystems. Majc architecture tutorial. <http://www.sun.com/microelectronics/MAJC/documentation/docs/majctutorial.pdf>, 1999.
- [49] Darek Mihocka. Pentium 4: In depth. <http://www.emulators.com/pentium4.htm>, 2001.
- [50] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Technical report, Cornell University, 1999.
- [51] Mozilla. How to debug memory leaks/refcnt leaks. <http://lxr.mozilla.org/seamoney/source/xpcom/doc/MemoryTools.html>, 2000.
- [52] Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled java executions. In *1999 ACM Java Grande Conference*, Palo Alt, CA, June 1999.
- [53] N. Nystrom and T. Hosking. Bloat: Bytecode level optimization and analysis tool, 1998.
- [54] Scott Oaks. *Java Security*. O’Reilly and Associates, 1998.
- [55] David A. Patterson and Carlo H. Sequin. Risc i: A reduced instruction set vlsi computer. In *Proceedings of the 8th International Symposium on computer architecture*, 1981.
- [56] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, 1995.
- [57] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity of operating systems. In *ICMAS Workshop on Immunity-Based Systems, 1996*, 1996.
- [58] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems 1*, 1:11–32, 1988.
- [59] G. Radin. The 801 minicomputer, 1982.
- [60] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace processors. In *30th International Symposium on Microarchitecture*, pages 14–23, December 1997.
- [61] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of java programs. In *European Conference on Object-Oriented Programming 1999*, 1999.
- [62] Robert S. Sielken. Application intrusion detection, 1999.
- [63] Mukesh Singhal and Niranjana G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw Hill Inc, New York, 1994.
- [64] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo ’00)*, Boston, MA, January 18 2000.

- [65] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 14–17 2000.
- [66] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [67] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.
- [68] B. Vasanth, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming, Language Design and Implementation*, 2000.
- [69] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.
- [70] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, 14(7):41–48, 1981.
- [71] Catherine Xiaolan Zhang, Zheng Wang, Nicholas C. Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26, 1997.