Safety Checking of Kernel Extensions

Craig Metz
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
`cmetz@inner.net`

Abstract

There are many places in operating systems today where extending the running kernel with small and fast extensions is an interesting thing to do. For example, the Berkeley Packet Filter (BPF) allows code for a virtual machine to be uploaded into a running kernel and executed at packet reception, allowing fairly arbitrary filtering of packets before they cross the expensive kernel to user interface. Whatever mechanism is used needs to provide some reasonable guarantees about the safety of the resulting code, which makes this problem complex.

This paper describes a simple x86 bytecode verifier that is intended to be used to verify that a small program that is to be loaded obeys a reasonable safety policy. For program constructs that it is able to reason about, it can verify that code does not execute privileged instructions, only accesses known memory locations, and terminates. It cannot reason about arbitrary programs, but can reason about simple programs and developers that know the prover's limitations can write their code to be recognizable by the verifier.

The contribution of this work is to show that a very limited prover can operate on native machine code and can efficiently reason about a small but still interesting set of programs.

1 Introduction

The Berkeley Packet Filter (BPF) is a simple virtual machine that is commonly found in the kernels of BSD-like systems. It allows user processes (that may or may not be privileged) to install simple extensions into the running kernel that are called at well-defined points. These extensions provide a service in kernel space on behalf of the user process that installed them, and provide greater flexibility than a data-only configuration interface can. The virtual machine has a fairly high overhead due to both the instruction set emulation and the run-time access checking, but this is much lower than the overhead of crossing into the user process.

Increasing network speeds and interest in applying this approach to other problems have made emulation overhead an issue. Also an issue is that the BPF program must be written in a special assembly language; no high-level language compilers currently exist that target the BPF machine. Native machine code has neither problem; it executes at full machine speed and can be targeted by whatever compilers are available. Many systems have a facility for extending the running kernel using native instructions (loadable kernel modules). Typically, these extensions are installed by a privileged process, are not subject to any sort of verification at installation time, and have the same run-time privileges as the rest of the kernel (in particular, there is usually almost no access checking). Such facilities are inappropriate for use by unprivileged processes, and represent serious risks even when restricted to privileged processes. In particular, the extension code can put the entire system in a bad state, either due to deliberate actions or unintended flaws.

A possible solution to the problem is to statically analyze program code to determine whether it obeys certain restrictions and safety policies. BPF already does this in a very limited way. The **`bpf_validate()`** function is called before code is installed into the running system, and checks three safety properties [1]:

1. "Check that jumps are forward, and within the code block."
2. "Check that memory operations use valid addresses" (within the VM)
3. "Check for constant division by 0"

At run-time, the BPF virtual machine provides memory access checking (by only allowing access through well-known pointers and bounds checking of offsets), provides no unsafe instructions, and guarantees that the C calling convention will be obeyed (because the virtual machine itself is implemented as a C function). Still, the run-time checking creates overhead.

A similar approach is used by the Java Virtual Machine (JVM), which uses a bytecode verifier to check for memory and type safety as code is loaded into the system[2]. Again, some safety is provided by the emulated machine, which provides no truly unsafe instructions and guarantees that the calling convention will be obeyed. However, the JVM approach alleviates the need for run-time bounds checking of memory accesses by moving that overhead to a load-time static analysis. While this improves run-time performance, there is still significant overhead due to the emulation of the JVM's instruction set.

In theory, the instructions present in the BPF virtual machine's instruction set and the JVM's instruction set are functionally equivalent to instructions (or short sequences of instructions) in native machine instructions. If it is possible to reason about the BPF or JVM instructions statically, it may be possible to do the same with a constrained version of a real machine's instruction set. This is an active area of research typically for use with micro-kernels, where fast and safe kernel extensions are more commonly needed. A particularly interesting research solution is Proof Carrying Code (PCC) [3], in which native code is statically analyzed, an attempt is made to generate a proof that the code obeys certain safety properties, and the proof and code are later verified by the kernel before accepting the code as an extension.

If a proof can be found and verified, then the code is known to have the safety properties and run-time checks are not needed [4]. If a proof cannot be found, this does not necessarily mean that the code is unsafe – as a nontrivial program property, this is an undecidable problem[5], but it is always a safe default action to consider code to be unsafe. If a programmer has the prover in hand and knows what the safety policy is, development can be done iteratively where the programmer adjusts the code in equivalent ways until it passes the prover; while the prover might not be able to reason about arbitrary programs, it is usually possible for a programmer to find a functionally similar program that it can reason about.

The main problem with the PCC approach is that the prover is too heavyweight, yet is still very limited in its ability to actually prove properties of programs. In one PCC implementation, the prover is a general-purpose theorem prover, which is very large and slow, but can attempt to prove rather arbitrary properties of programs. In practice, this approach becomes far less likely to successfully generate a proof as program complexity increases. In another, the prover is built into a special type-safe C compiler; most of the overhead of the prover is shared with the compiler and proofs are always generated for valid programs in the language (the property proved is type safety and the language is a type-safe language, so one is always possible), but now a specific compiler must be used. Because the proof generation step is so heavyweight, the PCC approach separates proof generation and proof verification and makes only the latter a trusted component.

A promising direction for a solution to this problem is to start with the PCC approach and to make the prover sufficiently fast and lightweight that the proof generation can be a trusted component (eliminating the need for an intermediate representation of the proof and a proof verifier). This is done by greatly constraining what programs are allowed to do; if programs are simple enough, reasoning about them becomes simple also. In particular, extensions are currently required to be stateless and return a simple integer result. This is still interesting for many problems such as packet filtering, and allows the extension to be terminated by the system without having to worry about cleanup. Conditions like division by zero are already checked at run-time "for free" by the hardware and global state can be recovered from rather painlessly, so there is little value in a relatively expensive static analysis to decide whether the condition is possible. In contrast, safety of memory accesses is expensive to do with the hardware (due to the time required to switch to more restrictive page tables), and backing out a write to a random memory location is painful if not effectively impossible, so statically checking for this kind of safety is valuable.

2 Prototype Prover

2.1 General Approach

The prover that was constructed takes native machine code and executes it using a simple simulator. The implementation currently only supports the x86 instruction set, though there is nothing in this approach that would prevent an implementation for another Instruction Set Architecture (ISA) (and most ISAs will actually be far easier to build an implementation for, but the popularity of the x86 instruction set motivated its use in this prototype). Privileged machine instructions and machine instructions that would not be output by a normal linear-mode x86 compiler (e.g., media instructions and instructions using segmentation) are not allowed in extensions. Floating point is also not allowed because hardware floating point instruction support may not be present on the machine and is not automatically emulated for code inside many kernels. Known values are tracked through the simulated execution of the program using a simple linear representation:

$$value = base + a \times x + b, x = 0..x_{max}$$

Where *base* is either 0 or an abstract unknown base variable whose concrete value will be supplied by the runtime environment (such as the initial value of the stack pointer, or the pointer to the function's input structure). Preconditions, such as the input values, descriptions of the accessible memory objects (alignment, size, and permissions), and the C calling convention's stack layout, are set up with their values. All other memory and register values in the system are set to "undefined," and the simulation is run. At all return points, postconditions such as register and stack restoration are checked.

Memory accesses are required to be to known values. Given a known value as above and memory objects in a similar representation, it is possible to check whether a memory access will always be in a defined region for which the program has permission to read or write. Known values are tracked on write, so that register spills to the stack do not lose information.

The simplifying assumption that makes the simulator function is that it is always safe to declare a value to be unknown if a known value cannot be easily computed. For example, most bitwise logical instructions yield a result of unknown because the simulator cannot create a linear function representation of their output given a linear function representation of their input. The main exception to this is bitwise AND, which is treated as constraining the linear function's range. This works in practice because most memory accesses are done through linear functions (scale-index-base, as seen in the x86 ISA), so the linear values tracked by the simulator contain enough information to reason about most memory addresses, while the other values used by a program aren't usually important for memory access safety.

The end result is a prover that can take short binaries compiled with `gcc` and verify their safety. The verified code can then be installed into a running kernel and used for small extension functions in the same way that the BPF virtual machine is used. These functions can be called as C functions through a front-end function, and are guaranteed to return and not to write except where explicitly permitted.

2.2 Implementation Walk-Through

In order to make discussion of the implementation more concrete, Figures 1-4 give an example of a short real-world filter program that was used extensively in the development of the prototype prover. First, a short libpcap filter program shown in Figure 1 was used as a starting point. It was chosen to be a short example of how BPF is typically used. The tcpdump program was used as a front-end to libpcap's internal compiler and optimizer, which generated the BPF program shown in Figure 2.

```
ip host 127.0.0.1 and udp port 42
```

Figure 1: Original libpcap filter program syntax

```
(000) ld      [0]
(001) jeq     #0x2000000        jt 2    jf 16
(002) ld      [16]
(003) jeq     #0x7f000001       jt 6    jf 4
(004) ld      [20]
(005) jeq     #0x7f000001       jt 6    jf 16
(006) ldb     [13]
(007) jeq     #0x11             jt 8    jf 16
(008) ldh     [10]
(009) jset    #0x1fff           jt 16   jf 10
(010) ldxb    4*([4]&0xf)
(011) ldh     [x + 4]
(012) jeq     #0x2a             jt 15   jf 13
(013) ldh     [x + 6]
(014) jeq     #0x2a             jt 15   jf 16
(015) ret     #1576
(016) ret     #0
```

Figure 2: BPF program resulting from compilation

Then the C program shown in Figure 3 was constructed by hand, attempting to be as faithful to the contents of the BPF program as possible. However, the C implementation uses abstractions for protocol headers and address information for the sake of readability and to show that programs that might be written by a real programmer compile down to analyzable code. The C program adds some run-time bounds checking that is not present in the BPF instructions – the BPF virtual machine does these checks implicitly, while a native program must do them explicitly.

Finally, the C program was compiled with the GNU C compiler into a native x86 program whose assembly listing is shown in Figure 4.

The first thing that the prover does is load the code of the program to be proven into a memory buffer. This requires the program to parse the i386 ELF binary object format, which is currently done in a reasonable but minimal fashion. Only the `.text` segment is loaded, which is assumed to contain exactly one C function, and no relocation is performed. The program loader is currently linked into the same binary as the prover, but is otherwise decoupled from it. In practice, the loader would be separated from the prover and would be an untrusted component (for example, a part of the user process).

Next, the prover initializes preconditions. The register pre- and post-conditions are listed in Figure 5, and are generic to the x86 C calling convention. The first four general purpose registers are initially set to

```
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>

struct bpf_preamble {
        uint32_t a;
};

int f(caddr_t p, unsigned int len)
{
        struct bpf_preamble *bpf_preamble;
        struct ip *ip;
        struct udphdr *udp;

        if (len < sizeof(struct bpf_preamble))
                goto fail;

        bpf_preamble = (struct bpf_preamble *)p;
        if (bpf_preamble->a != PF_INET)
                goto fail;

        if (len < sizeof(struct bpf_preamble)
            + sizeof(struct ip))
                goto fail;

        ip = (struct ip *)(p + sizeof(struct bpf_preamble));

        if (ip->ip_hl < 5)
                goto fail;
        if ((ip->ip_src.s_addr != htonl(INADDR_LOOPBACK)) &&
            (ip->ip_dst.s_addr != htonl(INADDR_LOOPBACK)))
                goto fail;
        if (ip->ip_off & IP_OFFMASK)
                goto fail;

        if (len < sizeof(struct bpf_preamble) + ip->ip_hl * sizeof(uint32_t) +
            sizeof(struct udphdr))
                goto fail;

        udp = (struct udphdr *)(p + sizeof(struct bpf_preamble) +
            ip->ip_hl * sizeof(uint32_t));
        if ((udp->uh_sport != htons(42)) &&
            (udp->uh_dport != htons(42)))
                goto fail;

        return 1576;

fail:
        return 0;
}
```

Figure 3: Equivalent C source code

```
.text
        .align 4
.globl f
        .type    f,@function
f:
        pushl %ebp
        movl %esp,%ebp
        pushl %esi
        movl 8(%ebp),%ecx
        movl 12(%ebp),%esi
        cmpl $3,%esi
        jbe .L8
        cmpl $2,(%ecx)
        jne .L8
        cmpl $23,%esi
        jbe .L8
        leal 4(%ecx),%edx
        movzbl 4(%ecx),%eax
        andl $15,%eax
        cmpl $4,%eax
        jle .L8
        cmpl $16777343,12(%edx)
        je .L18
        cmpl $16777343,16(%edx)
        jne .L8
.L18:
        testl $8191,6(%edx)
        jne .L8
        leal 0(,%eax,4),%edx
        leal 12(%edx),%eax
        cmpl %eax,%esi
        jb .L8
        leal 4(%ecx,%edx),%eax
        cmpw $10752,(%eax)
        je .L25
        cmpw $10752,2(%eax)
        jne .L8
.L25:
        movl $1576,%eax
        jmp .L26
        .p2align 4,,7
.L8:
        xorl %eax,%eax
.L26:
        popl %esi
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    f,.Lfe1-f
```

Figure 4: x86 program resulting from compilation (GCC 2.95.2, -O6)

"undefined," which is a special value in the prover that indicates that these cannot be read from until changed to a defined value (to prevent unknown or unintended values from being available to the program). The last four general purpose registers are initially set to abstract variables representing their initial value. This allows the program to generate values as offsets from their initial values (such as stack pointer relative addresses, which need to be resolved to an address relative to the initial value of the stack pointer) and it also allows the prover to check as a postcondition that the initial value has been faithfully restored when the program returns.

The memory preconditions are listed in Figure 6, and are specific to the problem of a BPF filter function with the function signature seen in Figure 3. Different functions will require different memory preconditions. The current prover implementation allows these to be changed easily. Memory words are defined to contain each input parameter, and the input packet buffer that is passed to the program is also defined. The input packet buffer is currently defined as a fixed-length buffer in order to greatly simplify the prover's operation; it is much easier to reason about falling within a known bound than an unknown bound (though this would be a useful feature to add in the future). At run-time, the caller would need to copy packets into a buffer of 8192 bytes in order for this to be safe. The input parameters are all treated as read-only because there is no legitimate need for them to be modifiable. In order to have some available scratch space, twelve words of stack space is set aside for read/write local variables. This scratch space is easily increased by defining more such words in the prover, but arbitrary amounts of temporary space are not supported. There are no memory postconditions; values that are read-only are never writable and thus need no postcondition checking, and values that are read-write are considered mutable.

Register	Precondition	Postcondition
`%eax - %edx`	(undefined)	n/c
`%esp`	sp_0	sp_0
`%ebp`	bp_0	bp_0
`%esi`	si_0	si_0
`%edi`	di_0	di_0

Figure 5: Register Pre/postconditions for the x86 C Calling Convention

Name	Base Variable	Offset	Size	Contents	Permissions
arg 0: `caddr_t p` at 4(%esp)	sp_0	+4	4	p	read
arg 1: `unsigned int len` at 8(%esp)	sp_0	+8	4	len	read
locals 0..11	sp_0	-4..-52	4	(undefined)	read/write
Input packet (`p`)	p	0	8192	(unknown)	read

Figure 6: Memory Preconditions for a BPF-like Filter Program

2.3 Limited x86 Simulation in the Prover

The core of the prover implements a very limited simulation of an x86 processor. The intent of this simulation is not to run the program or yield results, but to quickly attempt reason about the instructions and values used in a program and to attempt to determine whether memory accesses are safe.

A surprisingly complex part of this process is decoding the x86 instructions into the actual operation and operands. Figure 7 shows the general form of a x86 instruction. The x86 is a CISC architecture and earns the designation "complex." There are multiple opcodes for the same actual instruction, and the operands are determined by the opcode in ways that are frequently not well patterned. The instructions appear to be designed for a table-driven instruction decoder. For performance reasons, actual x86 chips probably implement this as optimized combinational logic, but development tools I looked at consistently used the table-driven decoding approach.

Appendix A of Intel's ISA reference[6] contains "opcode maps," which are tables of the mapping between opcode byte values and the instructions and operands they represent. Tools for the x86 instruction set commonly adapt these tables to drive their decoding of the instruction set, so this approach was chosen for use in the prover. The general structure of the tables works well for an implementation. There were many errors in the tables provided in Intel's documentation; in most cases, these errors were obvious when read, but there were a few cases that were more subtle and caused problems that were found during debugging of the prover. I submitted the first few corrections to Intel's developer support group and have yet to receive any response. Given that many of these errors are in parts of the tables that have not changed in over twenty years, this is unfortunate.

Figure 8 lists the subset of the Intel operand codes that are currently supported by the prover. Operands relating to segmentation, floating point, media instructions, and privileged or machine control instructions are all unsupported because the instructions that might use them are unsupported. Also unsupported are operand types not generated by `gcc`. These operand codes are used directly in the decode tables of

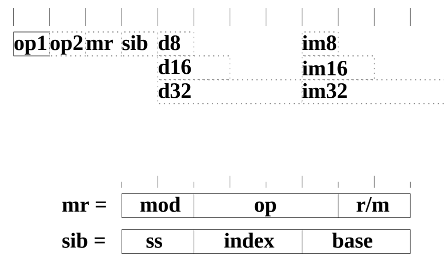

Figure 7: General Form of a x86 Instruction

the prover, along with a function pointer that calls a function that actually implements the operation specified.

An example of the resulting decode structure is the entry for opcode 0x89:

```
{ insn_mov, OP_Ev, OP_Gv, OP_NONE, FL_NONE },
```

This structure indicates that the actual operation is performed by the function `insn_mov()`, that its first parameter is a variable-length (that is, determined by the current word size mode) general purpose register, that its second parameter is a variable-length general purpose register or memory location, that this function has no third parameter (which most don't), and that this operation does not affect the `%eflags` register.

A set of important special cases are bytecodes that are not opcodes at all, and instead are prefixes that modify the following bytecodes. Most of these fall into categories of instructions that are not allowed (e.g., segment prefixes, address-size prefix, and the `lock` prefix), but two that need to be supported are the word-size prefix (0x66) and the two-byte opcode prefix (0x0f). Both are handled as if they were no-operand opcodes, and the functions that implement them return special values that are accumulated into an internal set of flags in the instruction decoder. These flags are accumulated and affect decoding of subsequent bytes until an "ordinary" instruction completion, at which point they are reset. One flag, used to implement the two-byte opcodes, switches the table used to decode opcode bytes. The other flag, used to implement the word-size prefix, causes a temporary operand structure to be created for the current instruction in which variable-length operand specifications are replaced with 16-bit length operand specifi-

Symbol(s)	Description
Ev/Ew/Eb	G.P. register, variable/16-bit/8-bit length
Gv/Gw/Gb	G.P. register or memory, variable/16-bit/8-bit length
Iv/Id/Iw/Ib	Immediate, variable/32-bit/16-bit/8-bit length
eAX/AL/AH	`%eax`, variable length, lowest 8 bits, next 8 bits
eCX/CL/CH	`%ecx`, variable length, lowest 8 bits, next 8 bits
eDX/DL/DH	`%edx`, variable length, lowest 8 bits, next 8 bits
eBX/BL/BH	`%ebx`, variable length, lowest 8 bits, next 8 bits
eSP/eBP	`%esp/%ebp`, variable length
eSI/eDI	`%esi/%edi`, variable length
CONST	constant operand (based on opcode) (non-Intel symbol)

Figure 8: Operand Types Currently Implemented

cations (thus changing them from 32-bit operands to 16-bit operands).

Instruction execution is simulated through short functions. These functions are not meant to model the actual execution of instructions. They are meant to track whether their results are predictable or not, and, if so, attempt to predict or put a range on the output values. So, for example, the `mov` instruction simply copies its first operand to its second; its second operand therefore has the same predictability properties as its first. In contrast, the `or` instruction takes two values and does a bitwise operation on them. If the two values are exactly known, an exactly known result can be generated; otherwise, the result of a bitwise `or` operation on at least one unknown quantity with a known quantity is difficult to reason about, and is simply considered to yield an unknown result. This is not a significant problem in practice because bitwise operations are not commonly used to generate addresses, and the prover is primarily interested in tracking values that are used as addresses.

One particularly tricky problem is conditional branch instructions. The prover was originally intended to use a reasonably sophisticated technique of tracking where result flags values came from and using some of those sources together with conditional branches to constrain values. For example, in order to implement reasoning about variable-length input blocks, the prover needs to be able to look at a comparison with the length variable and to separate what code is executed if the length is greater than or less than the value compared against. Otherwise, the code could be correctly checking for length and not doing accesses beyond the buffer, but the prover wouldn't be able to determine that. A simpler technique turns out to be surprisingly successful; all conditional branches are treated as a nondeterministic branch, and both forks are checked independently. Combined with a large fixed input buffer length, the inability to reason about values is not a problem. This also captures the critical property that run-time conditionals are typically not certain in advance (otherwise there would be no need for such a conditional), but that both sides of the branch must be to code that will do something safe.

Levels of nondeterminism and the total number of instructions that can be executed in a path are bounded in the prover. This is not so much designed to bound the program (though that is a consequence) as it is to bound the run-time of the prover. The intended applications of this prover are places where speed and small memory usage are important, and must be rather tightly bounded. The current prover allows up to 32 levels of nondeterminism and up to 128 instructions per path. These limits allow short and simple programs to be proven, which is the intent of the prover, but are not enough to allow complex programs to be proven safe, which probably would be outside the capacity of the prover anyway. These limits are very easily increased at compile-time if needed. Note that nondeterministic branches currently cannot rejoin; they become separate from every other branch until they terminate.

2.4 Preliminary Results

Preliminary results show that the extension code is sped up by an order of magnitude by being written in optimized native code rather than emulated BPF code. For the example used earlier, described in Figures 1-4, executed on a 333MHz Celeron CPU and entirely in-cache, the native code took an average of $0.130\mu s$, while the BPF code took an average of $1.328\mu s$. This is consistent with the PCC project's results and with this project's expectations. This is the recurring cost and is the cost that executes synchronously with the performance-critical part of the system (packet reception from the network in real-time), so an order of magnitude speedup is very helpful.

The more important result for this project is the performance of the prover itself. Related works, such as PCC, did not appear to make available solid information about the performance of their provers, but consistently indicated that their speed and memory consumption were fairly high (with the justification that that was a one-time cost). In order to be practical for inclusion into an operating system's kernel, the run-time and memory requirements for a prover must still be reasonably small. On the example used, the prover developed for this project took an average of $938\mu s$ of execution time, and required 29,964 bytes of heap space above 17,152 bytes of program (with a small amount of stack space also required). This is for a rough, un-optimized implementation. All of those requirements could probably be decreased linearly, but not by an order of magnitude. Implementing more complex reasoning in the prover might increase those costs.

3 Conclusions

This technique provides an alternative to BPF for packet filtering at gigabit speeds. In this application, an order of magnitude run-time performance increase is very helpful because receiving the packets is already pushing the limits of what common x86 hardware can do. With some straightforward modifications, the commonly used packet capture library libpcap can be made to generate native machine code rather than BPF code, which can then be passed to the kernel for verification and installation. Such an arrangement would allow many common packet-capture programs, such as tcpdump, ethereal, and the ISC DHCP server, to take advantage of this performance increase simply by linking with the new library. Alternately, libpcap could be completely bypassed and a compiler could be used to generate an optimized binary for installation in the kernel, but this would require applications to change.

The single greatest danger of this approach is that the prover is a trusted component and has not been developed using high-assurance software techniques. If the prover were to be flawed in such a way that an unsafe program could be loaded as an extension, it could circumvent the security of the entire system. The prover's implementation needs to be subjected to serious scrutiny before it could be deployed for use by unprivileged users. A reasonable strategy would be to initially constrain it to use by privileged users only.

Another problem is that the linear representation of values is done as the sum of several machine integers, each of which has as much precision as the sum will have in the running program. It may be possible to use this difference in precision to subvert the prover. A work-around that is currently employed is to constrain the range of values allowed in each field of a known value, so that sign reversals are not possible as long as the base values are not close to the beginning or the end of the number space.

For the immediately intended application, static verification appears to be viable and the prover implemented for this project appears to be a successful proof of concept. This technique is probably also applicable to other problems, but care must be taken to address the generality and trust issues of building a practical prover.

There has been a good bit of similar work to this, both on verification of synthetic machine codes that make properties easier to reason about at a higher runtime cost and on extensive verification of real machine codes at a high proof cost. The expected contribution of this work is to show that a very limited prover can operate on native machine code (to get the best performance for the code that's executing) and can efficiently reason about a very small but still interesting set of programs.

4 Acknowledgments

This work started out as a project for a special topics seminar on microprocessor architectures taught by Kevin Skadron of the University of Virginia.

Niels Provos, Kevin Skadron, and Chris Telfer reviewed early drafts of this paper and provided valuable feedback.

References

[1] U. C. Berkeley CSRG. bpf_filter.c. From 4.4BSD-Lite.

[2] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[3] George C. Necula and Peter Lee. Proof-Carrying Code. *CMU CS Tech Report CMU-CS-96-165*, September 1996.

[4] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time checking. *Proceedings of the USENIX Second Symposium on Operating Systems Design and Implementation*, October 1996.

[5] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[6] Intel Corp. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. 1999.