Ksplice: Automatic Rebootless Kernel Updates

Jeff Arnold and M. Frans Kaashoek
Massachusetts Institute of Technology
{jbarnold, kaashoek}@mit.edu

Abstract

Ksplice allows system administrators to apply patches to their operating system kernels without rebooting. Unlike previous hot update systems, Ksplice operates at the object code layer, which allows Ksplice to transform many traditional source code patches into hot updates with little or no programmer involvement. In the common case that a patch does not change the semantics of persistent data structures, Ksplice can create a hot update without a programmer writing any new code.

Security patches are one compelling application of hot updates. An evaluation involving all significant x86-32 Linux security patches from May 2005 to May 2008 finds that most security patches—56 of 64—require no new code to be performed as a Ksplice update. In other words, Ksplice can correct 88% of the Linux kernel vulnerabilities from this interval without the need for rebooting and without writing any new code.

If a programmer writes a small amount of new code to assist with the remaining patches (about 17 lines per patch, on average), then Ksplice can apply all 64 of the security patches from this interval without rebooting.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms Design, Reliability, Security

Keywords hot updates, dynamic software updates

1. Introduction

Contemporary operating systems regularly release kernel patches to repair security vulnerabilities. Applying these patches typically requires rebooting the kernel, which results in downtime and loss of state (e.g., all network connections). Even when computer redundancy is available, rebooting can lead to momentary interruption or cause unexpected complications, which means that reboots are commonly specially scheduled and supervised. Since rebooting is disruptive, many system administrators delay performing these updates, despite the greatly increased security risk—more than 90% of attacks exploit known vulnerabilities [Wang 2004]. This paper describes and evaluates *Ksplice*, a system for using traditional source code patches to construct *hot updates*, which change the running kernel.

The key novelty of Ksplice is that it prepares hot updates at the object code level instead of the source code level, which allows Ksplice to perform hot updates with minimal programmer involvement. Existing hot update practices, described in Section 7, rely on a programmer to write source code files with certain properties (e.g., [Chen 2006, Makris 2007]) or require manual inspection of the running binary to achieve safety guarantees and resolve ambiguous symbols (e.g., [Altekar 2005]) . Significant programmer involvement in creating a hot update increases both the cost and the risk of the update, which discourages the adoption of hot updates. Ksplice therefore performs hot updates for *legacy binaries* (unmodified binaries created without foresight of the update system) based entirely on existing information, such as a source code patch.

Ksplice analyzes the original kernel and the traditional source code patch by comparing compiled code (and its metadata) rather than source code. Ksplice does so to avoid implementation limitations, safety problems, and programmer involvement that go along with hot update systems that compare and rewrite source code to update legacy binaries. For example, determining how a patch changes a piece of software by finding differences at the object code level makes it possible to take into account all of the linguistic features of the target programming languages (for the Linux kernel, C and assembly) without creating special cases.

Constructing hot updates at the object code level, however, presents new design challenges related to a compiler's freedoms in compiling source code to object code. In particular, optimized object code can hide the intent of a source code patch and include undesirable code changes. Addressing the challenges of operating at the object code level requires carefully tracking compiler guarantees, which reveals safety and practicality problems with existing source-level hot update systems. Ksplice solves these challenges using two novel object code level techniques. First, Ksplice uses *pre-post differencing* to generate object code for the update. Second, Ksplice resolves symbols correctly and provides safety using *run-pre matching*.

We implemented Ksplice for Linux, but the Ksplice techniques apply to other operating systems and to user space

applications. To evaluate Ksplice, we applied Ksplice to 64 Linux patches for kernel security vulnerabilities from May 2005 to May 2008. The 64 include all documented x86-32 Linux kernel vulnerabilities from this time interval with greater consequences than denial of service. Of the 64 patches, 56 can be applied by Ksplice without writing any new code, and the remaining patches can be applied by Ksplice if a programmer writes a small amount of new code (about 17 lines per patch, on average).

The contributions of this paper are a new approach for constructing hot updates, two new techniques to realize this approach, and an evaluation against Linux security patches. To the best of our knowledge, Ksplice is the first hot update system to work on the object code level and the Ksplice evaluation is the first evaluation of any hot update system against a comprehensive list of the significant security vulnerabilities within a commodity kernel over a period of time. This evaluation demonstrates that hot update systems currently have the potential to eliminate all kernel security reboots, while requiring little additional work from any programmer.

The rest of this paper is organized as follows: The next section provides a brief overview of Ksplice's design. Section 3 and Section 4 describe the two techniques and how Ksplice uses them to construct hot updates at the object code level. Section 5 discusses considerations specific to the Ksplice implementation for Linux. Section 6 tests Ksplice against security patches from May 2005 to May 2008. Section 7 relates Ksplice to previous work. Section 8 summarizes our conclusions and directions for future work.

2. Design Overview

Many source code patches can be transformed into hot updates without a programmer writing any new code. Specifically, a patch alone provides sufficient information for a hot update when the patch does not make *semantic changes* to the kernel's persistent data structures—that is, changes that would require existing instances of kernel data structures to be transformed (e.g., a patch that adds a field to a data structure would require existing instances of that data structure to change).

Ksplice requires a programmer to check whether the target patch makes any semantic changes to data structures. Performing this check requires only seconds or a few minutes for most security patches. If a patch does make semantic changes, then that update can still be applied using Ksplice, but doing so will require a programmer to write some new code.

If the target patch does not make semantic changes, then no programming is necessary. Since kernel security patches and other important bug corrections are usually designed to change the software as little as possible, we can expect many of these patches to make no semantic changes to data structures. Our evaluation confirms this intuition for Linux security patches.

Since Ksplice is designed to update legacy binaries, it does not require any preparation before the system is originally booted. The running kernel does not need to have been specially compiled, for example.

When applying an update using Ksplice, normal operation of the system is only interrupted for about 0.7 milliseconds, which is negligible for most systems—and far shorter than any reboot. Even more importantly, the operating system's state is not disrupted when applying a Ksplice update, so network connections and open applications are not lost.

Ksplice performs replacements on entire functions; if any code within a function is modified by the patch, then Ksplice will replace the entire function. Ksplice replaces a function by linking a new version of the function, called the *replacement code*, into the kernel and by placing a jump instruction in the running kernel's memory, at the start of the obsolete function, to direct execution to the replacement code.

The performance impact of applying a Ksplice update is minimal. A small amount of memory will be expended to store the replacement code, and calls to the replaced functions will take a few cycles longer because of the inserted jump instructions.

Ksplice constructs updates at the object code layer, which helps it solve three challenges faced by hot update systems:

- generating the replacement code
- resolving symbols in the replacement code
- verifying the safety of an update

By looking in detail at these three aspects of the Ksplice design, we will clarify the advantages of focusing on the object code layer.

During our design discussion, we make few assumptions about the operating system. We assume kernel support for dynamically-loadable kernel modules, and, in order to make some examples more specific, we use terminology from the Executable and Linkable object code format (ELF) [TIS 1993], which is widely used by Linux, BSD, and Solaris. The ideas apply to any operating system or to user space applications. The described solutions require access to the source code of the to-be-updated application, so updates for proprietary software would need to be generated by its vendor or some other party with access to the software source code.

3. Generating replacement code using *pre-post* differencing

To generate replacement code, the hot update system must identify what changed after applying a patch and generate code for the differences. Ksplice addresses these tasks using a technique that we call *pre-post* differencing. Ksplice identifies the code modified by applying a patch by comparing the kernel's *binary* code before applying a patch (*pre*) with the kernel's *binary* code after applying the patch (*post*).

This section explains Ksplice's motivation for operating at the object code layer, a challenge with that approach, and how Ksplice addresses that challenge using *pre-post* differencing.

3.1 Motivation and challenge

One advantage of operating at the object code layer is that the code changes implied by a patch are readily apparent at the object code layer. Consider a patch that changes a data type in a function prototype in a C header file (e.g., from an `int` to a `long long`). Because of implicit casting, this patch implies changes to the executable code of any functions that call the prototyped function. By operating at the object code layer, Ksplice detects these changes without needing any information about the semantics of implicit casting in C. In contrast, if one operates at the source code layer, one would find that the callers of the prototyped function have not had their source code modified at all, even after C preprocessing.

Looking for object code differences does not require special cases in order to deal with language-level nuances such as implicit casting, function signatures, static local variables, and whether code is written in C or assembly. The object code differences, unlike the source code changes, already express what we actually care about—how and where the machine's execution might be changed.

Unfortunately, it is not obvious how one should construct replacement code corresponding to a source code patch while operating entirely at the object code layer—in other words, by looking at compiler output rather than the source-level contents of the patch. Compiler output can obscure the desired changes and how those changes can be separated from the rest of the compilation unit or optimization unit[1].

As an example, the GNU C compiler (gcc) will, by default, lay out an entire object file's executable text within a single section named `.text`, and the compiler will generate much code within this section that performs relative jumps to other addresses within this section. If a single function is changed in length by the source code patch, then many relative jump offsets throughout the entire object file might change as a result of what was originally a simple change to a single function. A hot update system that operates at the object code layer will need to find a way to deal with this complication.

3.2 Solution

In order to understand the effect of a source code patch on the kernel, Ksplice performs two kernel builds and looks at how the resulting object files differ (see Figure 1). First, Ksplice builds the original kernel source to create object files that we call the *pre* object files. Next, Ksplice applies the source code patch and performs another build, which

[1] For compilers such as `gcc` that never perform any optimizations across compilation unit boundaries, optimization units are the same as compilation units. For compilers that optimize an entire program at once, the entire program is one optimization unit.

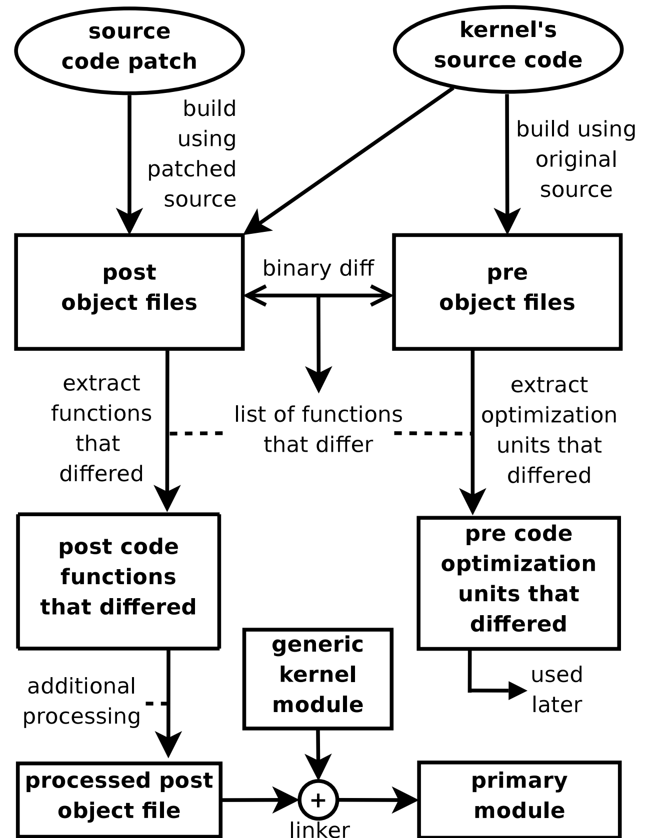

Figure 1: Generating replacement code based on a patch (Part I of generating a Ksplice hot update)

will recompile any object files whose constituent source files have been changed by the source code patch. We call the object files produced from this build the *post* object files. Ksplice compares the *pre* and *post* object code in order to determine which functions were changed by the patch.

In order to be practical, this comparison needs to avoid detecting extraneous differences between the *pre* and *post* object files. Without taking any special measures, these object files will contain many differences that are only tangentially related to the source code patch. In order to identify a smaller set of changes caused by the patch, Ksplice would like to be able to generate object code that makes no assumptions in its executable text about where functions and data structures are positioned in memory. Avoiding these layout assumptions is also important for creating the replacement code itself, since the replacement code needs to reference existing functions and data structures in the running kernel.

To reduce location assumptions, Ksplice's kernel builds are performed with certain compiler options enabled to ensure that every C function and data structure within the kernel receives its own section within the resulting object files. These options are known as `-ffunction-sections` and

-fdata-sections; they are included in several C compilers, including GNU's C compiler and Intel's C compiler, and they could be added to any compiler that does not include them. Enabling these options forces the compiler to generate relocations for functions and data structures, which results in more general code that does not make assumptions about where functions and data structures are located in memory. Instead, the resulting object code contains more general machine instructions along with relocation entries so that arbitrary addresses can be plugged-in later.

When compiling with these options, kernel functions that have not been changed directly by the source code patch will often have identical *pre* and *post* object code. For various reasons, such as nonlocal compiler optimizations, some of the resulting object code sections could differ in places not directly caused by the source code patch, but extraneous differences between the *pre* and the *post* object code are harmless. Although Ksplice would like to replace as few functions as possible, we can safely replace a function with a different binary representation of the same source code, even if doing so is unnecessary. In contrast, code differences between the running kernel object code and the *pre* object code could be harmful, but these differences are checked for using *run-pre* matching, which is discussed in Section 4.

After Ksplice has completed the comparison between the *pre* and *post* object code, it extracts the changed functions from the *post* object code and, after some processing, puts them into their own object file. Ksplice then creates a kernel module, known as the *primary module*, whose purpose is to load this processed *post* code into the kernel.

At this point, the replacement code in the primary module is not ready for execution because the symbols referenced by its relocations have not been resolved to memory addresses.

4. *run-pre* matching

This section introduces *run-pre* matching, which addresses two problems: how to resolve symbols in the replacement code and how to verify the safety of an update (if an update would be unsafe, *run-pre* matching will abort the update). We first discuss the challenges in addressing the two problems and then present *run-pre* matching.

4.1 Challenge of resolving symbols

Any hot update system needs to resolve symbols into addresses in order to fulfill relocations in the replacement code.

A simple way to resolve symbols is to use an available symbol table, such as the Linux kernel's kallsyms symbol table. Unfortunately, attempting to resolve symbols based on entries in a symbol table will commonly cause problems when a symbol name appears more than once or does not appear at all. For example, as in the patch for CVE-2007-0958, the replacement code might reference a symbol by the name "notesize", and two or more local symbols with that name might appear in the kernel. In this situation, the hot update system needs a way of determining which "notesize" address should be used to fulfill the relocation.

Given only the C source code of a running program and its complete symbol table, it is impossible to determine which "notesize" address corresponds to a particular "notesize" symbol in the source code. A source-level hot update system therefore cannot handle these patches—or at least cannot do so without laborious programmer intervention. By working on the object code level and using *run-pre* matching, Ksplice can handle this challenge.

4.2 Challenge of verifying safety

Hot update systems need to operate correctly even if the compiler performs optimizations that sometimes result in different object code being produced for the same source code. Compilers are allowed to make a wide variety of optimization decisions within an optimization unit, such as whether to inline a function at each of the locations that the function is called.

Consider the situation in which we want to replace a function obsolete_func with a new version of that function. We cannot determine where obsolete_func has been inlined in the running kernel by looking at any piece of source code. Since compilers commonly inline functions that do not have the inline keyword, this concern is not limited to some small subset of functions that say inline in the source.

Failing to replace obsolete_func in some of the places that it has been inlined could lead to serious problems such as data corruption (if, for example, obsolete_func's internal locking changed). Therefore, source code comparisons are not sufficient for a legacy binary hot update system to guarantee safety when replacing part of an optimization unit.

The only way to determine where obsolete_func has been inlined in the running kernel is to take into account the object code of the running kernel. Determining where obsolete_func has been inlined is still not straightforward since this information does not appear explicitly in the executable code, and compiler optimizations make it difficult to deduce through disassembly.

In addition to protecting against the problems that can arise from compiler optimizations, a hot update system should also try to protect against other dangerous conditions, such as a user providing "original" source code that does not actually correspond to the running kernel.

4.3 Solution: *run-pre* matching

Ksplice uses a single technique, *run-pre* matching, to address all of the above challenges.

We refer to the code in the running kernel as the *run* code. The safety verification problem arises when a hot update system makes unchecked assumptions about the *run* code (about where a function is inlined, for example). Since Ksplice already detects all differences between the *pre* code and the the *post* code, the remaining dangerous assumption is that the *pre* code is identical to the *run* code.

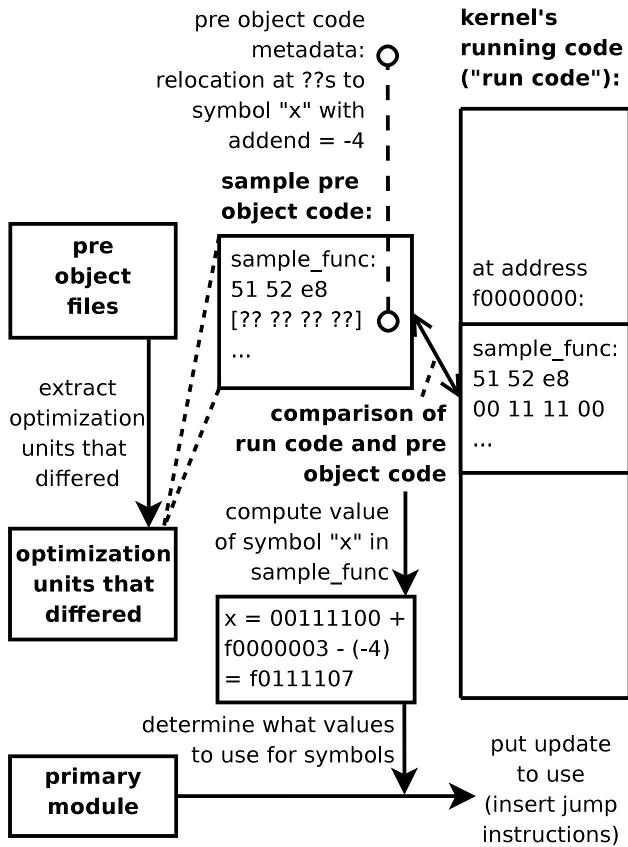

Figure 2: Determining symbol values via *run-pre* matching (Part II of generating a Ksplice hot update)

Ksplice can check that the running kernel's code meets its expectations by adding a step to the hot update process to check the *run* code against the *pre* code. Specifically, we should be concerned if we can find a difference between the *run* code and the *pre* code in the kernel optimization units that are being modified by the hot update.

During the process of comparing the *run* code against the *pre* code, the hot update system can also gain the symbol information that it needs to solve the symbol resolution challenge. The *run* code contains all of the information needed to complete the relocations for the *pre* code.

run-pre matching passes over every byte of the *pre* code, making sure that the *pre* code corresponds to the *run* code. When this process comes to a *pre* word of memory that is unknown because of a *pre* relocation entry with an ambiguous symbol name, Ksplice can compute the correct final *pre* address based on the corresponding *run* bytes in memory.

For example, consider the situation depicted in Figure 2 in which the *pre* code contains a relocation to a symbol called `x`. Assume that two or more local symbols called `x` appear in the kernel. The *pre* code generated by the compiler will, as in all relocation situations, not contain a final address for `x` at the to-be-relocated position. Instead, the *pre* code's metadata will know that a symbol name (`x`) and an "addend"[2] value are associated with that to-be-relocated position in the *pre* code. The to-be-relocated position's final value in memory will be computed from the addend (A), the value (S) for the symbol `x`, and the final address (P) of the to-be-relocated position. Specifically, this position will take on the value $A+S-P$.

When *run-pre* matching gets to the to-be-relocated location in the *pre* code, it will note that this relocation has not yet been fulfilled, and it will examine the *run* code in order to gain the information needed to fulfill it. The *run* code contains the already-relocated value *val*, which is $val = A+S-P_{run}$. The *run-pre* matching system also knows the *run* address of that position in memory (P_{run}). The *pre* code metadata contains the addend A, and so the symbol value can be computed as $S = val + P_{run} - A$.

In the example in Figure 2, the already-relocated value in the *run* code is `00111100`, so $val = \texttt{00111100}$. The *run* address of that position in memory is $\texttt{f0000000}+3$, so $P_{run} = \texttt{f0000003}$. The addend in the pre code metadata is -4, so $A = -4$. The symbol value for `x` in `sample_func` can therefore be computed as $x = \texttt{00111100} + \texttt{f0000003} - (-4) = \texttt{f0111107}$.

Ksplice does not strictly require that the hot update be prepared using exactly the same compiler version, assembler version, and compiler options that were used to prepare the original binary kernel, but doing so is advisable since the *run-pre* check will, in order to be safe, abort the upgrade if it detects unexpected object code differences. Obtaining exactly the same complier version and assembler version that were used to prepare the original binary kernel is straightforward, so we do not consider in detail exactly how often *run-pre* matching will abort when a slightly different compiler version or assembler version is utilized.

Since `-ffunction-sections` puts each function into its own section, small relative jump instructions can turn into longer jump instructions when `-function-sections` is enabled. As a result, in order to operate correctly, the *run-pre* matching system needs some architecture-specific pieces of information. First, the matching system must be able to recognize no-op instruction sequences on that architecture. In order to manipulate code alignment, assemblers will sometimes insert efficient sequences of machine instructions that are equivalent to a no-op sequence. The *run-pre* matching system needs to be able to recognize these sequences so that they can be skipped during the *run-pre* matching process.

Second, *run-pre* matching must know basic information about the instruction set, such as the lengths of all instructions and the list of instructions that take an offset that is rel-

[2] The "addend" is an offset chosen by the compiler to affect the final to-be-stored value. For x86 32-bit relative jumps, this value tends to be -4 to account for the fact that the x86 relative jump instructions expect an offset that is relative to the starting address of the *next* instruction.

ative to the program counter, such as jump instructions. This information can be obtained from a disassembler for the architecture. The matching system needs this information so that it can verify that relative jumps in the *run* and the *pre* code point to corresponding locations even though they use different relative jump offsets.

5. Implementation

We implemented Ksplice for Linux 2.6 on the x86-32 and x86-64 architectures. Although small parts of Ksplice, such as the jump instruction assembly code, need to be implemented separately for each architecture, most of the system is architecture-independent.

Ksplice provides a command `ksplice-create` that takes as input the original kernel source and a patch in the standard patch format, the unified diff patch format. `ksplice-create` writes an update file which can then be applied to the kernel using `ksplice-apply` and reversed using `ksplice-undo` (reversing an update removes the jump instructions so that the original function text is once again executed). Here is an example of creating an update for the `prctl` vulnerability, CVE-2006-2451, using a patch file called `prctl` and a kernel source directory `~/src`:

```
user:~$ ksplice-create --patch=prctl ~/src
Ksplice update tarball written to ksplice-8c4o6u.tar.gz
[user then becomes the superuser]
root:/home/user# ksplice-apply ./ksplice-8c4o6u.tar.gz
Done!
```

After executing these two commands, the running kernel has been updated without a reboot.

5.1 Components

Ksplice's implementation consists of four components:

- a Ksplice core kernel module, written in C, responsible for performing *run-pre* matching, inserting the jump instructions, and other kernel space responsibilities
- user space software, written in C and Perl, that uses the input to generate the processed *pre* and *post* object files, with help from the GNU object code library, BFD [Chamberlain 1991]
- a trivial kernel module for loading the *pre* code, called the *helper* kernel module
- a trivial kernel module for loading the *post* code, called the *primary* kernel module

The user space software links the processed *pre* and *post* object files into the *helper* and *primary* kernel modules before those modules are loaded into the kernel. The *helper* and *primary* kernel modules register themselves with the Ksplice core kernel module when they are loaded. After an update has been applied, its *helper* module can be unloaded in order to save memory. Since the *helper* module must contain the entire optimization unit corresponding to each patched function, it can be much larger than the *primary* module.

run-pre matching is performed in kernel space, rather than in user space, because some systems do not trust the user space administrator user with the ability to modify the kernel. On these systems, the kernel will only load modules that have been cryptographically signed. *run-pre* matching produces trusted symbol information, so it cannot be performed in user space without trusting the user space administrator with the ability to modify the kernel.

5.2 Capturing the CPUs to update safely

A safe time to update a function is when no thread's instruction pointer falls within that function's text in memory and when no thread's kernel stack contains a return address within that function's text in memory.

Ksplice uses Linux's `stop_machine` facility to achieve an appropriate opportunity to check the above safety condition for every function being replaced. When invoked, `stop_machine` simultaneously captures all of the CPUs on the system and runs a desired function on a single CPU.

If the above safety condition is not initially satisfied, then Ksplice tries again after a short delay. If multiple such attempts are unsuccessful, then Ksplice abandons the upgrade attempt and reports the failure.

Ksplice's current implementation therefore cannot be used to automatically upgrade *non-quiescent* kernel functions. A function is considered non-quiescent if that function is always on the call stack of some thread within the kernel. For example, the primary Linux scheduler function, `schedule`, is non-quiescent since sleeping threads block in the scheduler. This limitation does not prevent Ksplice from handling any of the significant Linux security vulnerabilities from May 2005 to May 2008.

Ksplice's call to `stop_machine` takes about 0.7 milliseconds to execute. During part of that time, other threads cannot be scheduled on the system, but this scheduling delay is acceptable for most systems.

5.3 Using custom code to assist an update

Ksplice allows a programmer to write custom code to be called from within the kernel during the update process. This custom code can modify data structures atomically at the time that the jump instructions are inserted, as is required by patches that make semantic changes to data structures.

The programmer can, by modifying the source code patch, add custom code to any compilation unit(s) within the kernel. The custom code can access any functions and variables that are normally accessible to code in that scope.

For example, if the programmer wants to call a new function called `myupdate` at the time that the update is applied, the programmer simply needs to modify the source code patch so that it does the following:

- adds the function called `myupdate` to any compilation unit in the kernel
- includes the special Ksplice header "`ksplice-patch.h`" in that compilation unit
- adds the macro call "`ksplice_apply(myupdate);`" to that compilation unit

The `ksplice_apply` macro writes a function pointer to a special section in the resulting object file. This special section instructs Ksplice to call the target function when Ksplice has the machine stopped for inserting the jump instructions. Ksplice also provides macros called `ksplice_pre_apply` and `ksplice_post_apply` for performing setup and cleanup that does not need to happen while the machine is stopped. Three analogous macros are provided for calling functions when an update is reversed.

Other than the macros described above, Ksplice does not special-case this custom code in any way; its replacement code is generated using the same Ksplice machinery as any code that is added by a patch.

5.4 Patching a previously-patched kernel

When one wants to apply a new patch to a previously-patched running kernel, Ksplice needs to be provided with two inputs, which are similar to the standard Ksplice inputs:

- the source for the currently-running kernel, including any patches that have been hot-applied (this source is called the "previously-patched source")
- the new source patch (which should be a difference between the previously-patched source and the desired new source)

The update process is almost exactly the same as before. The *pre* object code is generated from the previously-patched source code, and the *post* object code is generated from the previously-patched source code with the new patch applied. The *run-pre* matching system will compare *pre* object code against the latest Ksplice replacement code already in the kernel.

6. Evaluation

In this evaluation, we are interested in how many Linux kernel security patches can be applied without writing any new code and how many lines of code are needed to apply the remaining patches. This section describes the security patches, the method used for evaluating when a hot update is successful, and Ksplice's results.

6.1 Linux security patches

We compiled a list of significant Linux 2.6 kernel security problems from May 2005 to May 2008 [Arnold 2008]. We compiled this list of vulnerabilities and the corresponding patches by starting from the complete Common Vulnerabilities and Exposures (CVE) vulnerability list [MITRE 2008] and, using the CVE entry descriptions and other metadata, matching all of the Linux kernel entries on that list against the Git source control logs of the Linux kernel (specifically, Linus Torvalds' kernel tree [Torvalds 2008] and the unified stable kernel trees [Kroah-Hartman 2008]). We included on our Linux kernel vulnerability list only security problems that could result in greater consequences than denial of service; specifically, all of the vulnerabilities on the list involve the potential for some kind of privilege escalation (about two-thirds) or information disclosure (about one-third). We excluded from the list any architecture-specific vulnerabilities that do not affect the x86-32 architecture.

Figure 3: Number of patches by patch length

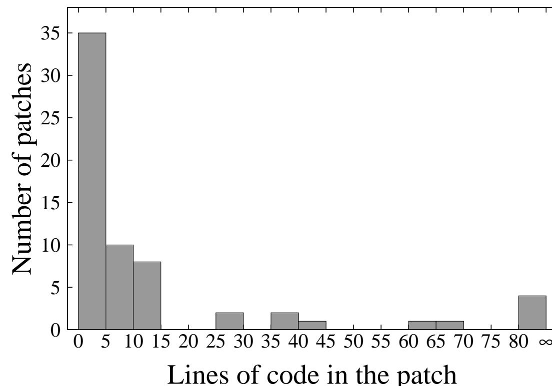

Figure 3 shows that most Linux kernel security vulnerabilities can be corrected by modifying relatively few lines of source code. Of the 64 vulnerabilities from this time interval, 53 vulnerabilities were corrected in 15 or fewer lines of source code changes, and 35 vulnerabilities were corrected in 5 or fewer lines of changes.

Reviewing the text of these patches reveals that most Linux kernel vulnerabilities can be corrected without making any semantic changes to persistent data structures.

6.2 Method

To evaluate Ksplice, we applied Ksplice to the 64 patches described in Section 6.1. We are interested in how many security patches from this interval can be applied successfully to running kernels. In this evaluation, success was judged by three criteria. First, the update needed to apply cleanly; more specifically, (a) *run-pre* matching needed to observe no inconsistencies between the *run* and *pre* object code, (b) all symbols in the replacement code needed to be resolved, and (c) the stack check needed to pass. Second, the kernel needed to continue functioning without any observed problems while running a correctness-checking POSIX stress test [Waterland 2007]. Third, for the vulnerabilities for which exploit code was available, we also tested that the exploit code worked before the hot update and did not work after the hot update.

Since no single Linux kernel version needs all 64 of the security patches (many of the vulnerabilities were introduced during the three year period of ongoing development), we tested Ksplice with the 64 patches using six different Linux kernels released by the Debian GNU/Linux distribution and eight different "vanilla" Linux kernels released by kernel.org on behalf of Linus Torvalds. The list of kernels used in the evaluation and the details of which patches were tested on each kernel are available online [Arnold 2008].

The vanilla kernels were only introduced into the evaluation in order to test patches that are not applicable to any released Debian kernel. For most of the patches in this category, the problems corrected by the patch were completely absent from any released Debian kernel. Some vulnerabilities are caught before they make it into released kernels seen by users, and some vulnerabilities affect portions of the kernel that are completely disabled by Linux distributors.

We obtained the original binary and source Debian kernel packages from an archive of nearly all packages released by Debian since mid-2005 [Ukai 2008]. For each kernel, we began by fetching the compiler and assembler versions originally used by Debian in order to compile that binary kernel. We then used the Debian kernel source for that kernel as input to Ksplice, along with an unmodified security patch taken directly from the kernel trees described in Section 6.1. In order to perform the hot update on a running machine, we installed the corresponding binary Debian kernel package on a machine and booted that kernel.

6.3 Results

We have used Ksplice to correct all 64 of the significant 32-bit x86 kernel vulnerabilities during the time interval. 56 of the 64 patches can be applied by Ksplice without writing any new code. The remaining eight patches require a programmer to write 17 new lines of code each, on average.

The eight patches that require new code are shown in Table 1, along with the required number of new logical lines (semicolon-terminated lines) of C code. The patches in this table require new code because, by changing how a data structure is initialized or by adding a field to a data structure, these patches change the semantics of persistent data structures in the kernel. Some of the patches that change the initial value of a data structure do so by explicitly changing the C variable declaration, but most of them do so by modifying a data structure initialization function.

The new code generally needs to perform a single task, such as walking a linked list to update all existing instances of a data structure. The full text of the new code is available online [Arnold 2008].

For every recent x86 vulnerability for which we could locate working exploit code, we confirmed that the exploit stops working when the corresponding hot update is applied. Specifically, we have performed this verification for CVE-2006-2451 [Hernandez 2006], CVE-2006-3626 [R. 2006], CVE-2007-4573 [Elhage 2007], and CVE-2008-0600 [qaaz 2008].

Table 1: Patches that cannot be applied without new code

CVE ID	Patch ID	Reason for failure	New code
2008-0007	`2f98735`	changes data init	34 lines
2007-4571	`ccec6e2`	changes data init	10 lines
2007-3851	`21f1628`	changes data init	1 line
2006-5753	`be6aab0`	changes data init	1 line
2006-2071	`b78b6af`	changes data init	14 lines
2006-1056	`7466f9e`	changes data init	4 lines
2005-3179	`c075814`	changes data init	20 lines
2005-2709	`330d57f`	adds field to struct	48 lines

Ksplice would be unable to safely achieve these results without operating at the object code layer. First, because Ksplice generates replacement code at the object code layer, it supports two kinds of patches that have never been supported by an automatic source-level hot update system—changes to function interfaces and changes to functions that contain static local variables. In the evaluation, this capability was needed for 8 of the 64 patches.

Second, as described in Section 3.1 and Section 4.2, source-level hot update systems for legacy binaries do not handle inline functions properly. Function inlining cannot be ignored as uncommon; 20 of the 64 patches from the evaluation modify a function that has been inlined in the *run* code, despite the fact that only 4 of the 64 patches modify a function that is explicitly declared `inline`.

Third, *run-pre* matching allows Ksplice to resolve ambiguous symbol names in situations that would be impossible for a source-level hot update system for legacy binaries. The Linux 2.6.27 code enabled in Debian's default kernel configuration contains 6,164 symbols that share their name with other symbols in the kernel—that's 7.9% of the total number of symbols in the kernel. The symbols with ambiguous names are spread throughout the kernel source; 21.1% of the compilation units in the kernel contain at least one such symbol. Five of the 64 patches from the evaluation modify a function that contains a symbol with an ambiguous name.

As a specific example from the evaluation, consider correcting the security vulnerability CVE-2005-4639 on Linux kernel 2.6.16. The patch for this vulnerability changes the function `ca_get_slot_info` in the file `dst_ca.c`. This function accesses a variable called "`debug`", which is a global static variable in this file. Unfortunately, the file `dst.c` also contains a static global variable called "`debug`". A source-level hot update system working from a symbol table would not be able to determine which address in the kernel corresponds to each "`debug`" symbol.

As a final example of the benefits of an object-level approach, consider the patch for CVE-2007-4573. This patch modifies the x86-64 assembly file `ia32entry.S` in order to

zero-extend all registers in order to avoid an arbitrary execution vulnerability in the 32-bit kernel entry path. Ksplice handles this patch using the same techniques and code that handle patches to pure C functions. A source-level hot update system for C would not be able to deal with the pure assembly files in the kernel.

None of the original binary kernels used in the evaluation had `-ffunction-sections` or `-fdata-sections` enabled, but *run-pre* matching always succeeded because these options only result in the limited object code differences discussed at the end of Section 4.3.

7. Related Work

There are two streams of work related to Ksplice: academic papers and black hat publications. We discuss the relationship of Ksplice to these in turn.

7.1 Research literature

A key difference between Ksplice and previous systems is that Ksplice operates at the object code layer. Hot update systems that are designed to update legacy binaries—specifically, LUCOS [Chen 2006], OPUS [Altekar 2005], and DynAMOS [Makris 2007]—construct updates at the source code layer, which results in the design limitations discussed in Section 3 and Section 4. Ksplice does not have these limitations because it operates at the object code layer.

Because of the complexity of analyzing a patch and constructing the replacement code at the source code layer, LUCOS and DynAMOS leave this process to a kernel programmer. In these systems, a programmer needs to construct replacement source code files with special properties, which requires "tedious engineering effort" [Chen 2006] and, as with any significant human involvement, is error-prone.

LUCOS is virtualization-based kernel hot update system which requires a customized version of the Xen virtual machine monitor. LUCOS uses the virtual machine monitor in order to gain a high degree of control over the kernel during the update process. By controlling the kernel's underlying hardware, LUCOS can, for example, intervene when particular addresses in memory are accessed. Unlike LUCOS, Ksplice does not require virtualization.

OPUS is a user space hot update utility for C programs that targets security updates. OPUS requires the least programmer work of any of the previous hot update systems, but OPUS cannot handle function signature changes, changes to functions with static local variables, and changes to assembly files. Also, a programmer using OPUS must perform a manual check for inline functions in the to-be-updated binary in order to ensure patch safety (looking for the `inline` keyword in the source code is not sufficient since compilers routinely inline functions that lack the keyword). Ksplice's design avoids these problems by approaching hot updates from the object code layer.

Even though Ksplice's overall approach is different from previous work, Ksplice shares specific techniques with previous work. For example, Ksplice uses DynAMOS's "shadow data structures" method for adding a field to a data structure [Makris 2007]. Like DynAMOS, Ksplice provides functions for helping a programmer to utilize shadow data structures. In the Ksplice evaluation, we used this capability in order to apply one of the patches, CVE-2005-2709.

DynAMOS also describes a technique for a programmer to manually update a non-quiescent kernel function. Ksplice's hooks for running custom code during the update process, described in Section 5.3, allow a programmer to use the DynAMOS method for updating non-quiescent kernel threads. Since safely performing this technique requires programmer knowledge of the code in question, both DynAMOS and Ksplice leave this process to a programmer.

The Ksplice evaluation is the first evaluation of a legacy binary hot update system against a comprehensive list of patches over a time interval. Ksplice's evaluation measures Ksplice against all 64 of the significant Linux x86-32 security vulnerabilities over a three year time interval. The DynAMOS and LUCOS evaluations each describe testing five patches; the OPUS evaluation describes testing 26 patches from a corpus of 883 user space vulnerabilities.

DynInst [Buck 2000], KernInst [Tamches 1999], and Arachne [Douence 2005] are systems for instrumenting running legacy C binaries. Unlike the hot update systems described above, these systems are not designed for updating the source code of a running program; they do not address the problem of converting a traditional piece of source code into a change to a running program. Instead, DynInst and Arachne each provide their own custom language for describing what operations should be inserted into the running program. Even if one could write a source-to-source compiler to convert traditional C source code patches into these custom languages, the resulting system would necessarily have the aforementioned disadvantages of performing hot updates on the source code layer.

In addition to the systems described above which can update legacy binaries, many other research systems have been created for modifying specially-designed running programs. We now discuss some of these systems.

The K42 research operating system has implemented hot update capabilities in K42 [Baumann 2007; 2005] by relying on particular abstractions provided by that operating system's highly object-oriented kernel. However, "few systems include a uniform indirection layer" (or rely exclusively on the factory abstraction targeted by the K42 hot update system), which "would limit the applicability of [K42] dynamic update" to other software systems [Baumann 2005]. Also, since the K42 hot update system is structured around replacing K42 object classes, it cannot handle changes to any code outside of the object system, such as exception-handling code or other low-level code. Additionally, like DynAMOS

and LUCOS, the K42 hot update system leaves a programmer responsible for constructing replacement source code files based on a patch.

Ginseng [Neamtiu 2006] is a user space hot update utility for C programs that has been used to upgrade three open source server programs across several years' worth of releases. Unlike the hot update systems described above, Ginseng cannot be used to update legacy binaries because it requires significant changes to programs at the source code layer before they are originally compiled. In particular, Ginseng rewrites a program's C source code to support upgrades via function indirection and type wrapping, and Ginseng expects a programmer to annotate the to-be-updated software to indicate one or more safe update points. Additionally, a programmer may need to refactor some coding patterns. Ksplice does not require any code changes before the to-be-updated software is originally started and, in the common case, does not require programmer annotations or other new code. Ginseng remains the most extensively evaluated hot update system for upgrading realistic C programs across many full releases. Evaluation of Ksplice or other legacy binary hot update systems in this domain is future work.

The DAS operating system [Goullon 1978] included hot update primitives in the operating system, but these primitives could not be used to upgrade the kernel.

Gupta [1993] built an early system for performing hot updates on C user space programs that is a predecessor to OPUS. Unlike OPUS, the system requires programs to be linked against a special library, and, during an upgrade, it loses program state stored in the kernel because of how it creates a new process instead of modifying the old one.

Gupta [1996] proved that verifying whether or not a programmer has provided a correct transition function for accomplishing a source code upgrade is, in the general case, undecidable. In other words, a hot update system cannot, in the general case, prove that a patch, along with a state transition function, results in a valid state for the new program.

Lastly, researchers have looked at ways to upgrade a machine's operating system if all of the important applications on that machine support application-level migration between machines. For example, incoming web server traffic can be redirected from one machine to another to allow an operating system upgrade to occur. Using virtualization techniques, researchers have implemented this approach using multiple virtual machines on one physical machine [Lowell 2004]. This approach works only for applications that support migration between machines at the application level.

7.2 Black hat techniques

The black hat community has been performing hot updates on commodity operating system kernels for many years as part of rootkits. Computer attackers benefit from modifying the kernel so that they can hide their activities and exert a high level of control over the system.

Publications on rootkits for the Linux, BSD, and Windows operating systems [Cesare 1998, Hoglund 2005, Kong 2007, sd@sf.cz 2001] describe techniques that aid in the construction of hot updates for these platforms. Black hats, however, have different hot update goals than system administrators; a black hat only needs one manually-constructed hot update in order to succeed, and, in general, a black hat is more willing than a system administrator to tolerate a slight chance that a hot update will destabilize the target machine.

Instead of pursuing a generalized approach for safely accomplishing arbitrary hot updates, these documents tend to focus on simple approaches which usually work for accomplishing particular goals. For example, these publications suggest using memory patterns called "keys", which are a few bytes long, in order to find particular parts of the kernel, such as particular data structures, in memory. These multibyte keys can have several problems, such as appearing several times in memory or not appearing at all on a different machine. Various strategies exist for obtaining a "reasonable guess at how useful a key is and if a key is not at all stable" [Cesare 1998] (for example, a person can try a key on multiple machines and insert a wildcard into the key if necessary), but these strategies are laborious and do not provide much confidence about whether the update will work. Ksplice's approach for generating expectations for the contents of kernel memory and systematically mapping symbol names to values is significantly more general than the techniques described in these rootkit publications.

Although Ksplice is safer and easier to use than existing hot update practices, Ksplice does not provide malware authors with any troubling capabilities that they do not already possess. Black hats have known for many years how to create rootkits that accomplish their goals using ad hoc kernel inspection and modification techniques. For this reason, once an attacker has unrestricted access to kernel memory, a computer system must already be assumed to be completely compromised. The best way to protect against attackers is to promptly patch security vulnerabilities so that attackers never gain unrestricted access to kernel memory. The goal of Ksplice is to make this kernel patching process easier.

8. Conclusions and Future Work

We have presented a system, Ksplice, that takes as input a source-level patch and then constructs and applies the corresponding updates to a running kernel without rebooting it. Because Ksplice constructs hot updates at the object code level, Ksplice can apply updates with little programming effort. Ksplice uses two new object code level techniques (*pre-post differencing* and *run-pre matching*) to generate code, resolve symbols, and provide safety for hot updates. Using Ksplice's implementation for Linux, a system administrator can eliminate all reboots associated with Linux security updates, which is a notable advance over the current state.

One could use Ksplice to create hot update packages for common starting kernel configurations. People who subscribe their systems to these updates would be able to transparently receive kernel hot updates along with the user space software updates to their system. This kind of distribution of hot updates would, without any ongoing effort from users, significantly reduce how frequently they need to reboot for updates to take effect. Distribution of hot kernel updates can reduce downtime, decrease windows of security vulnerability, and improve the user experience.

Acknowledgments

We thank Tim Abbott and Anders Kaseorg for improvements to the Ksplice implementation and for assistance with the Ksplice evaluation; Tim and Anders wrote the security patch custom code and helped generate some of the statistics in Section 6.3. We thank Russ Cox, Nelson Elhage, Sam Hartman, Anders Kaseorg, Tim Abbott, and Robert Morris for discussions that influenced the direction of this work.

References

[Altekar 2005] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, pages 19–19, 2005.

[Arnold 2008] Jeff Arnold and M. Frans Kaashoek. Ksplice evaluation full data: kernel versions, commit ids, and new code, 2008. URL `http://www.ksplice.com/cve-evaluation-2008`.

[Baumann 2007] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2007.

[Baumann 2005] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 32–32, 2005.

[Buck 2000] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Journal of High-Performance Computing Applications*, 14(4):317–329, 2000.

[Cesare 1998] Silvio Cesare. Runtime kernel kmem patching, 1998. URL `http://doc.bughunter.net/rootkit-backdoor/kmem-patching.html`.

[Chamberlain 1991] Steve Chamberlain. Lib bfd, the binary file descriptor library, 1991. URL `http://sourceware.org/binutils/docs-2.19/bfd/index.html`.

[Chen 2006] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd ACM conference on Virtual Execution Environments*, pages 35–44, 2006.

[Douence 2005] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Proceedings of the 4th conference on Aspect-oriented Software Development*, pages 27–38, 2005.

[Elhage 2007] Nelson Elhage. Root exploit for cve-2007-4573, 2007. URL `http://web.mit.edu/nelhage/Public/cve-2007-4573.c`.

[Goullon 1978] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. *IEEE Transactions on Software Engineering*, 4(4):298–307, 1978.

[Gupta 1993] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software—Practice & Experience*, 23(9):949–964, 1993.

[Gupta 1996] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[Hernandez 2006] Roman Medina-Heigl Hernandez. Local r00t exploit for prctl core dump handling, 2006. URL `http://seclists.org/fulldisclosure/2006/Jul/0235.html`.

[Hoglund 2005] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005. ISBN 0321294319.

[Kong 2007] Joseph Kong. *Designing BSD Rootkits*. No Starch Press, 2007. ISBN 1593271425.

[Kroah-Hartman 2008] Greg Kroah-Hartman. Linux kernel unified stable trees, 2008. URL `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6-stable.git`.

[Lowell 2004] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *SIGPLAN Notices*, 39(11):211–223, 2004.

[Makris 2007] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM EuroSys Conference on Computer Systems*, pages 327–340, 2007.

[MITRE 2008] MITRE. Common vulnerabilities and exposures list, 2008. URL `http://cve.mitre.org/cve`.

[Neamtiu 2006] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM conference on Programming Language Design and Implementation*, pages 72–83, 2006.

[qaaz 2008] qaaz. Root exploit for cve-2008-0600, 2008. URL `http://milw0rm.com/exploits/5093`.

[R. 2006] Joanna R. Root exploit for cve-2006-3626, 2006. URL `http://milw0rm.com/exploits/2013`.

[sd@sf.cz 2001] sd@sf.cz and devik@cdi.cz. Linux on-the-fly kernel patching without lkm, 2001. URL `http://www.phrack.org/issues.html?issue=58&id=7#article`.

[Tamches 1999] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd symposium on Operating Systems Design and Implementation*, pages 117–130, 1999.

[TIS 1993] Tool Interface Standard TIS. Executable and linkable format specification, 1993. URL `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

[Torvalds 2008] Linus Torvalds. Linux kernel tree, 2008. URL `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git`.

[Ukai 2008] Fumitoshi Ukai. snapshot.debian.net, 2008. URL `http://snapshot.debian.net`.

[Wang 2004] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, pages 193–204, 2004.

[Waterland 2007] Amos Waterland. The stress workload generator for posix systems, 2007. URL `http://weather.ou.edu/~apw/projects/stress/`.