OPUS: Online Patches and Updates for Security

Gautam Altekar Ilya Bagrak Paul Burstein Andrew Schultz

{galtekar,ibagrak,burst,alschult}@cs.berkeley.edu
University of California, Berkeley

Abstract

We present OPUS, a tool for dynamic software patching capable of applying fixes to a C program at runtime. OPUS's primary goal is to enable application of security patches to interactive applications that are a frequent target of security exploits. By restricting the type of patches admitted by our system, we are able to significantly reduce any additional burden on the programmer beyond what would normally be required in developing and testing a conventional stop-and-restart patch. We hand-tested 26 real CERT [1] vulnerabilities, of which 22 were dynamically patched with our current OPUS prototype, doing so with negligible runtime overhead and no prior knowledge of the tool's existence on the patch programmer's part.

1 Introduction

Security holes are being discovered at an alarming rate: the CERT Coordination Center has reported a 2,099 percent rise in the number of security vulnerabilities reported from 1998 through 2002 [27]. The onslaught of security violations seems unlikely to abate in the near future as opportunistic hackers focus their attention on exploiting known application vulnerabilities [3].

Software patching is the traditional method for closing known application vulnerabilities. After a vulnerability has been disclosed, vendors expediently create and distribute reparative patches for their applications with the hope that administrators will download and install them on their systems. However, experience indicates that administrators often opt to delay installing patches and in some cases, to not install them at all [23].

Administrators forego patch installation for several reasons. At the very least, applying a conventional patch requires an application restart, if not a full system restart. For many applications, the resulting disruption is often too great [5]. Perhaps more problematic, patches have become extremely unreliable due to shortened testing cycles and developers' tendency to bundle security fixes with feature updates. These unreliable patches can seriously debilitate a system, leaving administrators with no reliable method of undoing the damage [23]. Nevertheless, the industry's recognition of the need for small, feature-less security patches [27] and renewed emphasis on patch testing [11] promises to mitigate irreversibility and unreliability concerns. But even with these practices in place, the call for disruption-free security patches remains unanswered.

We believe that the disruption imposed by conventional patching is and will continue to be a strong deterrent to the quick installation of security patches. While resource-rich organizations combat the disruption through the use of redundant hardware, rolling upgrades, and application-specific schemes [6], many home and small-business administrators lack similar resources and consequently, they are forced to tradeoff application work-flow for system security. As the number of issued security patches continues to increase, the lack of transparency inherent in conventional patching becomes more evident, serving only to further dissuade users from patching.

Home and small-business administrators need a non-disruptive alternative to conventional security patching. Toward that goal, we introduce OPUS–a system for constructing and applying *dynamic security patches*. Dynamic security patches are small, feature-less program fixes that can be applied to a program at runtime. Many types of vulnerabilities are amenable to dynamic patching: buffer overflow, format string, memory leaks, failed input checks, double frees, and even bugs in core application logic (e.g., a script interpreter bug). Our survey of CERT vulnerabilities over the past 3 years has confirmed this to be the case.

Applying dynamic patches to running programs introduces additional complexity and implies fewer guarantees of the patch's correctness. In fact, determining the correctness of dynamic patches has been proven undecidable in the general case [15]. OPUS addresses this theoretical limitation by supplying the programmer with warnings of program modifications that are *likely* to result in an unsafe dynamic patch. We derive these warnings from the static analysis of patched and unpatched code. Once the programmer is confident with the patch's safety, OPUS produces a dynamic patch that can be disseminated to end-users who in turn can immediately apply the patch using the OPUS patch application tool.

Despite expectations, our preliminary experience using OPUS on dozens of real security patches and real applications free of instrumentation reveals that the process is surprisingly safe and easy. We attribute this result primarily to the small, isolated, and feature-less nature of security patches, and secondarily to OPUS's support for the C programming language and seamless integration with a widely used development environment (GCC and the Make system). While we hypothesized that static analysis would significantly aid the programmer in constructing safer patches, security patches proved to be so simple in practice that programmer intuition often sufficed.

The rest of the paper is organized as follows. We first describe the basic design goals of OPUS in section 2. Then we describe the abstract patch model assumed by OPUS in section 3. Section 4 fleshes out all major components of the OPUS architecture, while section 5 relates noteworthy implementation challenges. Experience and evaluation are described in section 6. Related work in the general area of vulnerability defense is highlighted in section 7. Finally, we propose some future work in section 8 and conclude in section 9.

2 Design considerations

The key observation behind OPUS is that most security patches are small, isolated, and feature-less program fixes. This observation motivates our goal of building a widely applicable dynamic security patching system rather than a generic dynamic software upgrade mechanism. The following design decisions reflect the practical nature of our approach:

Support the C programming language. Given the significant amount of work done in type-safe dynamic software updating [10, 16], it would be ideal if programs were written in type-safe languages that preclude many security bugs. However, programmers have been reluctant in making the transition, often citing the cost of porting to a safe language as a justification for inaction. By supporting C as the ubiquitous unsafe programming language, OPUS is able to accommodate legacy code without monumental effort on the part of the programmer.

Integrate with existing development environment. No system is entirely practical if it intrudes on the software development and deployment infrastructure. OPUS works transparently with standard Unix build tools (GCC and Makefiles). Moreover, it integrates smoothly with large-scale software projects engineered with little foresight of our tool.

Estimate dynamic patch safety. In general, it is impossible to guarantee that dynamically and statically applied versions of the same program change will exhibit identical program behavior [15]. To mitigate (but not eliminate) the danger of producing an unsafe dynamic patch, OPUS employs static analysis to point out potentially dangerous program changes. In particular, we assume that unsafe dynamic patches arise from modification of state that is not local to the function being patched and provide warnings for all such instances.

Require no code annotations. Many existing dynamic update techniques provide programmer generated annotations or transition functions to assist in patch analysis and application (e.g., [16, 26]). While these systems are very flexible with respect to the types of patches they admit, the annotation features they provide are rarely necessary in the constrained domain of security patching. By not supporting such features in OPUS, we were able to simplify its design and enforce our policy of admitting only isolated and feature-less program modifications.

Patch at function granularity. In OPUS, the smallest possible patch still replaces a whole function definition. Patch modifications that spread over multiple functions will result in a patch containing the new definitions for all functions affected. The new code is invoked when control passes to the updated function's body. Our decision to patch at function granularity not only simplified reasoning about patch safety, but also eased implementation. The alternative, that of patching at sub-function granularity, is too cumbersome to implement and results in no appreciable benefit to the user.

3 Patch model

In abstract terms independent of the implementation specifics, the patch model is intended to answer two questions:

1. What kinds of patches don't work with OPUS?

2. What kinds of patches may not be safe when used with OPUS?

To answer the first question, we present detailed descriptions of inadmissible patches (i.e., patches that OPUS outright rejects). We then answer the second question by defining our notion of dynamic patch safety.

3.1 Inadmissible patches

Inadmissible patches are classified into two types: those that are prohibited due to fundamentally hard limitations of dynamic patching and those that are excluded to ease our initial implementation. While it is unlikely that we can overcome the fundamentally hard limitations, future versions of OPUS will eliminate many of the current implementation-related limitations.

3.1.1 Fundamental limitations

No patching top-level functions. A dynamic patch is useful only if the patched function is bound to be called again at some future point in the execution of the program. If top-level functions such as C's `main` are patched, the modifications will never take effect. This is also true for functions that run once before the patch is applied and, due to the structure of the program, will never run again.

No changes to initial values of globals. All globals are initialized when the program is loaded into memory. Thus, all initialization is complete before the patch is ever applied and as a result, modifications to global initial values will never take effect.

3.1.2 Implementation limitations

No function signature changes. In C, a function signature is defined by its name, the type of each of its arguments, and the type of its return value (argument name changes are allowed). A straightforward way to handle altered function signatures is to consider it as a newly added function and then patch all the functions invoking it. In practice, however, we rarely encountered security patches that alter function signatures and therefore, we decided not to support them in our initial implementation. Thus, all modifications introduced by a patch must be confined to function bodies.

No patching inlined functions. C supports explicit inlining through macro functions and GCC supports implicit inlining as an optimization. In principle, a patch for a macro function can be considered as a patch for all the invoking functions. However, it's more difficult to determine if GCC has implicitly inlined a function. Since we rarely encountered patched inline functions in our evaluation and because we wanted to keep our prototype implementation simple, we chose to prohibit inlining all together.

3.2 Patch safety

A dynamic patch is *safe* if and only if its application results in identical program execution to that of a statically applied version of the patch. More precisely, a safe patch does not violate the program invariants of any function other than the one in which the change was made.

Without programmer assistance, the problem of static invariant checking is undecidable. Therefore, OPUS adopts a conservative model of patch safety. We say that a patch is *conservatively safe* if and only if the function changes involved do not make additional writes to non-local program state and do not alter the outcome of the function's return value. By non-local state, we mean any program state other than that of the patched function's stack frame. Examples include global and heap data, data in another function's stack frame, files, and sockets.

A conservatively safe patch does not change the program invariants of unpatched functions (assuming a conventional application). Thus, conservative patch safety implies safety as defined above. The converse, however, is not true and therefore, a problem with the conservative safety model is that it is subject to false positives: it labels many patches as dangerous when they are in fact safe. For example, consider the following patch for BIND 8.2.2-P6's zone-transfer bug [29]:

```
*** 2195,2201 ***
      zp->z_origin, zp_finish.z_serial);
   }
   soa_cnt++;

   if ((methode == ISIXFR)
-->             || (soa_cnt > 2)) {
      return (result);
   }

} else {

*** 2195,2201 ***
      zp->z_origin, zp_finish.z_serial);
   }
   soa_cnt++;

   if ((methode == ISIXFR)
-->             || (soa_cnt >= 2)) {
      return (result);
   }

} else {
```

Although the patch is safe (as verified by the programmer), it is not conservatively safe due to its alteration of the return value: the function now returns `result` when `soa_cnt` equals 2. Unfortunately, most security patches alter the function return value in a similar manner, implying that strict adherence to the conservative patch model will preclude a majority of patches in our domain.

Since we believe (and have verified to some extent) that most security patches are safe in practice, a key goal of OPUS is to admit as many security patches as possible. Therefore, if a patch doesn't meet the conservative patch model, OPUS does not reject it. Rather, it informs the programmer about the violation, thereby allowing him or her to invoke intimate knowledge of the program before making the decision to accept or reject the patch.

4 Architecture

4.1 Overview

OPUS combines three processing stages: (1) patch analysis, (2) patch generation, and (3) patch application. The interconnection between these stages is governed by simple annotations that serve as a way to maintain intermediate state and hand off information from one stage to the next. This allows for greater flexibility in terms of the architecture's future evolution and keeps the pieces in relative isolation from each other, making them easier to compose. The high-level architecture is presented in Figure 1.

The OPUS architecture is motivated by two high-level usability requirements: (1) user interaction with the system should appear natural to a programmer, and (2) the system should fit seamlessly on top of an existing software build environment developed with *no* foresight of OPUS. We meet these design goals by augmenting the standard C compiler and substituting it for the default one, which simultaneously suits our analysis needs and preserves the native build environment of the patched application (assuming that it is normally built with the default compiler).

The programmer interacts with the OPUS front-end similar to the way a programmer would interact with a regular C compiler. Compile time errors are still handled by the compiler proper with OPUS generating additional compiler-like warnings and errors specific to the differences found in a given patch. The programmer invokes the annotation analysis on the patched source, acts on any warning or errors, and finally invokes the *patch generation* to compile the dynamic security patch. The *patch injector* then picks up the dynamic patch and applies it into a running process.

4.2 Annotations and interface languages

The preamble to the patch analysis is a script that performs a `diff` of the changed and the unchanged source trees and invokes the appropriate build target in the project's master Makefile to initiate the build process. Using the diff information, OPUS generates a series of `.opus` files, one in every directory that has a changed source file. The purpose of the `.opus` files is to notify the instrumented C compiler which files have changed and will require static analysis (described below).

The instrumented C compiler parses `.opus` files when invoked from the project's Makefile. If the file is present, then a new set of annotations is generated for the file by the compiler ("Source annotations" in Figure 1). The line ranges found in `.opus` annotations are useful as an aid to the programmer because they restrict the warnings and errors the patch analysis produces to only the line ranges associated with the patch (we obtain line ranges from the textual `diff`, but other more sophisticated approaches are also possible [17]). The policy is reasonable since only the changed lines are considered problematic as far as patch safety is concerned (see section 3); the unmodified code is assumed to work correctly.

Ideally, the old and the new source trees would be processed in parallel obviating the need for external state in the form of annotations. The established build environment, however, prevents us from invoking the compiler in the order most convenient to us. To circumvent this practical limitation, static analysis, which uses only the local (per file) information, is handled by the instrumented compiler, while the rest of the cross-tree analysis is deferred to the annotation analysis.

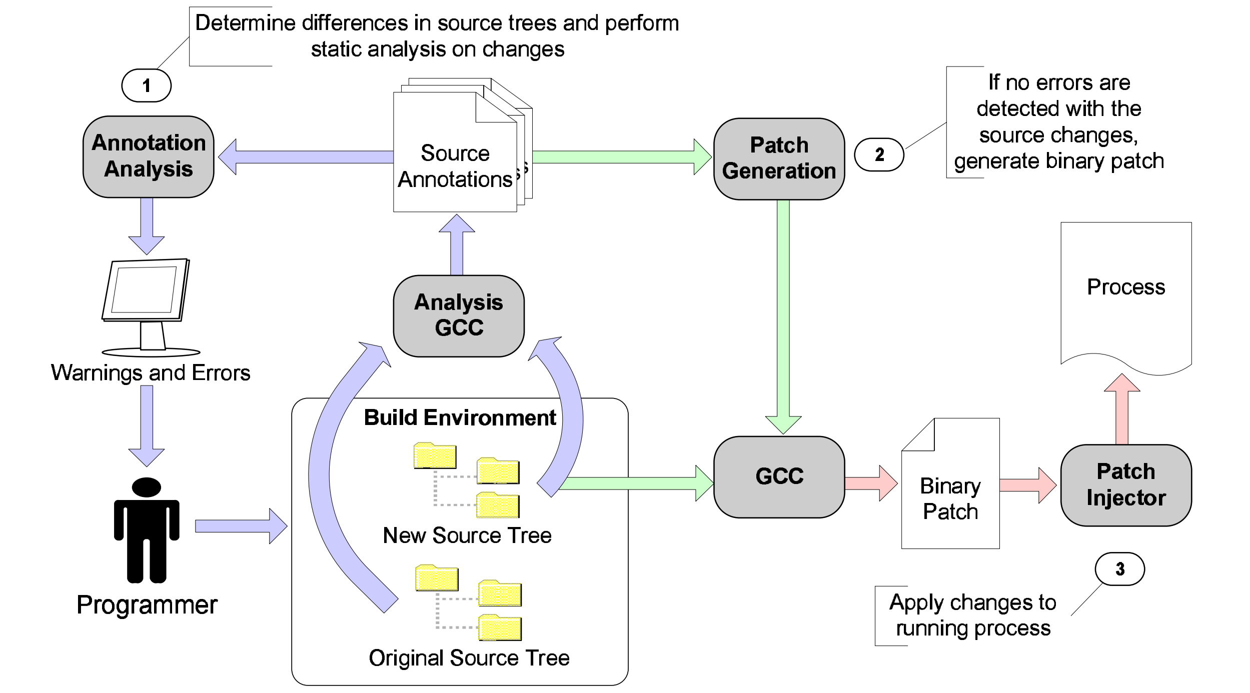
The source annotations produced by the instrumented compiler contain tags specifying which function definitions were changed by the patch. Similarly, the annotations include hashes of function prototypes for cross-tree comparison by our patch analysis. We also include in the annotations a list of globals so that addition of new globals is detected across the patched and unpatched source trees.

4.3 Static analysis

The static analysis portion of the system has two goals. One is to determine whether the patch is admissible as described in section 3.1. The other is to determine if the patch satisfies our definition of conservative safety described in section 3.2.

We address the first goal by generating source annotations that are fed into the annotation analysis, which then alerts the programmer if the patch meets one of the inadmissible criteria. For instance, if the annotations indicate that the signature for a given function has changed, an

Figure 1 OPUS high-level architecture



error message is returned to the user. The error message denotes explicit rejection of the patch.

We address the second goal using static analysis. The goal of static analysis is to direct the patch writer's attention to program changes likely to result in an unsafe dynamic patch as defined by our conservative safety model. The current implementation of static analysis checks for only one component of our conservative safety model. In particular, it checks if there are any new writes to non-local state (e.g., global variables, data in another function's stack frame) within the patched function, and if there are, it marks those writes as harmful. Our current implementation does not check for altered return values (the other component of the conservative safety model): such functionality requires program slicing, which we haven't fully implemented yet.

Given that our current implementation focuses only on identifying new writes to non-local state, the chief difficulty is that the set of C variables that can be referenced/dereferenced to affect the non-local state is a dynamic property of a program. To address this difficulty, we employ a conservative static analysis algorithm that computes a set of all local variables that *could* be used to point and write to data outside of the function's stack frame. In effect, our analysis is similar to the static analysis designed to catch format string vulnerabilities, except in the format string case the tainted set of expressions is the set whose values may have arrived over an untrusted network [25].

4.3.1 Bootstrapping static analysis

Success of the static analysis depends on a crude over-estimation of the set of variables determined to refer to non-local state. We call this set the *tainted set*, and the variables in it — *tainted variables*.

For any given function, the set of function arguments of pointer type (as determined by the `pointer_type_p` predicate) is considered tainted by default. Since neither the content nor origin of these pointers is known in general, we assume they point to non-local state. Currently, we do not perform inter-procedural static analysis. However, we would like the tainting to be conservative, and therefore the analysis also taints any pointer variable assigned as a return value of a callee.

4.3.2 Taint flow propagation rules

Figure 2 contains a subset of rules for computing the tainted set. In our notation $\mathbf{R}$ stands for a set of tainted expressions, and e is a string of C source denoting some expression. The statement "given a set of tainted expressions $\mathbf{R}$, running the taint flow algorithm on expression e results in a new set of tainted expressions $\mathbf{R}'$" is written as $\mathbf{R} \vdash e \Rightarrow \mathbf{R}'$.

FUNCTION rule captures our assumption in regard to pointer type function arguments. IF specifies that each branch of the *if* expression is unaffected by the expressions tainted in the other branch, but the statements following the entire expression are processed as if taintings from both branches have been applied. BINOP rules and SEQ show how taint sets are modified when analysis walks over an expression tree and a sequence of statements, respectively.

The ASSIGN family of rules cover the way the left hand side of an assignment statement gets tainted, with each rule specifying what happens when an array type, struct type and a pointer type variable is assigned. Finally, the base case EMPTY rule shows that an empty expression leaves the tainted set unaltered. We omit the rest of the rules for brevity, but they follow the same high-level pattern.

The static analysis generates warnings when it encounters the following situations in the program's source code: (1) a reference to non-local data is dereferenced on the left hand side of an assignment or (2) a new value is assigned to an explicitly-named global variable. The reader should be careful not to confuse the computation of the tainted set and the conditions under which a warning is issued. Specifically, aliasing of pointer variables produces no warnings whereas dereferencing a tainted pointer on the left-hand side of an assignment does.

4.3.3 Limitations of static analysis

Implementation limitations. Our current implementation of static analysis warns only about new modifications to non-local program state. A true conservative analysis, however, should also produce warnings for altered return values. This can be a problem, for example, in the following sorting function, which periodically invokes a comparator function to determine the desired ordering on the input data.

```
void sort() {
   qsort(array_of_numbers, array_length,
         sizeof(int), &comparator);
}

int comparator(int* a, int* b) {
    return *a > *b;
}
```

Suppose that the comparator function is modified such that the ordering is reversed as follows:

```
int comparator(int* a, int* b) {
    return *a < *b;
}
```

Further suppose that the program starts to sort with the old comparator function, but then is dynamically patched. Subsequently, it finishes the sort using the new comparator function. The resulting "sort" does not correspond to any sort produced by a statically applied version of the original or patched version on the same data.

The above patch eludes our current implementation of static analysis because it does not modify any non-local data. However, it indirectly violates program semantics through a change in return value and therefore violates our notion of conservative safety. Ideally, a warning should be produced, but our current implementation does not do so, implying that the programmer must consider the effects of the patch with respect to the return value.

Fundamental limitations. Ignoring the implementation limitations, a static analysis that strictly adheres to the conservative safety model will generate false warnings for many security patches (examples of which are given in section 3.2 and section 6.5). These false warnings have to be overridden by programmer intuition, which implies that OPUS introduces some programmer overhead in the patch development process. Perhaps more problematic, incorrect programmer intuition may result in an unsafe dynamic patch. In the end, OPUS can only alert the programmer to the potential dangers of dynamic patching. It cannot guarantee that a dynamic patch is equivalent to its static version nor can it point out flaws in the patch itself.

4.4 Patch generation

Once the programmer is satisfied with the patch, having removed any errors and examined any warnings generated by the analysis, the patch generation stage can be invoked. Although OPUS does patching at function granularity, a patch object is actually a collection of changed functions aggregated based on the source file of their origin.

The first step in the patch generation is to pin down exactly which files need to be compiled into a dynamic patch object. The generation system does this by parsing the annotation files sprinkled throughout the new and old project source trees by the instrumented compiler. The annotations are inspected in a pairwise fashion, identifying which functions or globals have been added and which functions have changed.

Next, the patch generator runs the source code through the C preprocessor to create a single file with all the header files spliced in. The static analysis tool is then reinvoked on the preprocessed and stripped source code, dumping annotation files which contain the new (post-preprocessed) line numbers for each function and global variable.

The final step is to cut and extern the preprocessed source. Cutting removes all of the code for any functions

Figure 2 Operational semantics for computing the set of tainted expressions.

$$\frac{}{\mathbf{R} \vdash e_{empty} \Rightarrow \mathbf{R}} \text{(EMPTY)} \qquad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1 = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{(ASSIGN)}$$

$$\frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1.field* = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{(ASSIGN-FIELD)} \qquad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1[*]* = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{(ASSIGN-ARRAY)}$$

$$\frac{v \in \mathbf{R} \; ; \; \texttt{pointer\_type\_p}(v) \; ; \; \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \oplus \quad e \Rightarrow \mathbf{R}' \cup \{\oplus \quad e\}} \text{(BINOP1)}$$

$$\frac{v \in \mathbf{R}' \; ; \; \texttt{pointer\_type\_p}(v) \; ; \; \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \oplus \quad v \Rightarrow \mathbf{R}' \cup \{\oplus \quad v\}} \text{(BINOP2)} \qquad \frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}' \; ; \; \mathbf{R}' \vdash e_2 \Rightarrow \mathbf{R}''}{\mathbf{R} \vdash e_1; e_2 \Rightarrow \mathbf{R}''} \text{(SEQ)}$$

$$\frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}_1 \; ; \; \mathbf{R}_1 \vdash e_2 \Rightarrow \mathbf{R}_2 \; ; \; \mathbf{R}_1 \vdash e_3 \Rightarrow \mathbf{R}_3 \; ; \; \mathbf{R}_2 \cup \mathbf{R}_3 \vdash e_4 \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash if \ (e_1) \ then \ \{e_2\} \ else \ \{e_3\} \ e_4 \Rightarrow \mathbf{R}'} \text{(IF)}$$

$$\frac{\begin{array}{c} \texttt{function\_type\_p}(e) \\ \{e.args[i] \mid 0 < i < e.numargs \wedge \texttt{pointer\_type\_p}(e.args[i])\} \\ \cup \ \{v \mid \texttt{global\_p}(v)\} \vdash e.body \Rightarrow \mathbf{R} \end{array}}{\{\} \vdash e \Rightarrow \mathbf{R}} \text{(FUNCTION)}$$

that have not changed by blanking the line ranges of the function definition. Externing involves placing an "extern" storage modifier before any function or global variable that is not new or changed. The end result is a single source file that contains code for only new or changed functions, and extern definitions for any other variables which have not changed, but references to which are needed for successful compilation.

Once the processed source is ready, the generation system invokes the standard C compiler on the code. Additionally, OPUS adds the `-shared` compiler switch which causes the compiler to create a shared object. When all of the shared objects have been compiled, OPUS packs them together in an archive with a patch definition file and an unstripped copy of the original program binary. The resulting archive comprises a dynamic patch object, which can then be transferred to the machine in need of patching and applied to a running process by the patch injector.

4.5 Patch application

The patch application process is straightforward and consists of two distinct phases. In the first phase, the patch installer attaches to a specified process. Once attached, the installer gains complete control over the process: it can inspect and modify the process's address space, intercept signals sent to the process, and can even execute code on the child's behalf.

In the second phase, the installer attempts to apply the patch by redirecting calls of the target functions to the newer versions contained in the patch. Before applying the patch, however, the patch installer must ensure the patch safety criteria discussed in section 3: current execution point cannot be under the dynamic scope of a target function, i.e., no frames on the stack should belong to the function being patched. If any of the stacks contain activation frames of any of the target functions, the safety criteria does not hold and patching is deferred.

Handling multiple threads posed a unique challenge in the design of the patch injector. It is possible, although unlikely, for threads to never exit the dynamic scope of a target function. In such a case, program execution will never satisfy our safety condition.

5 Implementation

A fully functional OPUS prototype has been developed and vetted on real examples of dynamic patches (see section 6). We now present noteworthy implementation challenges encountered while building an OPUS prototype based on the preceding architecture.

5.1 GCC integration

GCC version 3.4.2 was taken as a baseline for our implementation. The actual modifications to it were minimal — around 1,000 lines of code spread over 5 files. Modifying GCC directly has imposed several implementation challenges not the least one of which has been simply grokking GCC APIs and finding the right time in the compilation process to invoke our analysis. As a benefit, the static analysis effectively supports all features of the C programming language, including arcane C extensions supported by GCC [12, 18].

Despite some of the benefits of integration, one of our current action items is removing the static analysis from GCC and implementing it externally under a tool like *cil* [21]. We hope to report on the new version of the static analysis in the final version of the paper.

The critical aspect of the current implementation is that both the standard and the instrumented compilers produce identical answers on identical inputs. For any arbitrarily complex build environment where a default GCC is used, the modified version "just works" in its place.

5.1.1 L-values

The ASSIGN "family" of taint flow rules make the tainting of the left hand side of an assignment expression appear straightforward. In reality, C allows deeply structured l-values that may include complex pointer manipulation and conditionals, not just array indexes and structure field accesses [18].

Consider a contrived example of an assignment:

```
(a == 42 ? arr1 : arr2) [argc] = a;
```

In the example above, it cannot be determined statically which of the arrays gets tainted. When an anomalous l-value is encountered we alert the user and request that a statement be rewritten as an explicit conditional.

Similar problems can arise when processing left hand sides with array reference and the index expression swapped and anything but the most trivial pointer arithmetic. The examples that follow illustrate the non-trivial expressions that can appear as l-values.

```
argc[argv] = 42;
(arr1 + (arr2 - (arr3 + 1)))[0] = 42;
((int) argv + (char**) argc)[0] = NULL;
```

This class of non-trivial assignment statements actually requires some type-checking to disambiguate the target of the assignment. The type-checking piece turned out to be a great implementation hurdle in the GCC-integrated version of the analysis, and is one of the reasons we are considering a rewrite.

5.2 Patch injection up close

The patch installer can be thought of as a finite state machine with two states (see Figure 3): each state corresponding to the execution of either the child thread(s) or the installer thread. The installer periodically stops the execution of the child thread to determine if the thread is safe to be patched and if so, it moves on to the second stage of actually applying the patch atomically. Our criterion for safety (see section 3) is met via runtime stack inspection of the thread we attach to, while the `ptrace` system call is actually used to attach to the thread in the first place.

5.2.1 Patch setup

At the fist stop signal received from the child, the patch injector sets up for the patch. This involves gathering data on the functions that need to be changed, setting up a code playground that will be used to execute code on the behalf of the child, gathering information on all the threads that are running, and setting up the indirection table used to specify the new function addresses.

For each function, we obtain its starting address in the text segment as well as the code length via the `nm` and `objdump` commands. The starting address is used for inserting breakpoints at the beginning of a patched function and the code length is used at the stack inspection stage (see section 4).

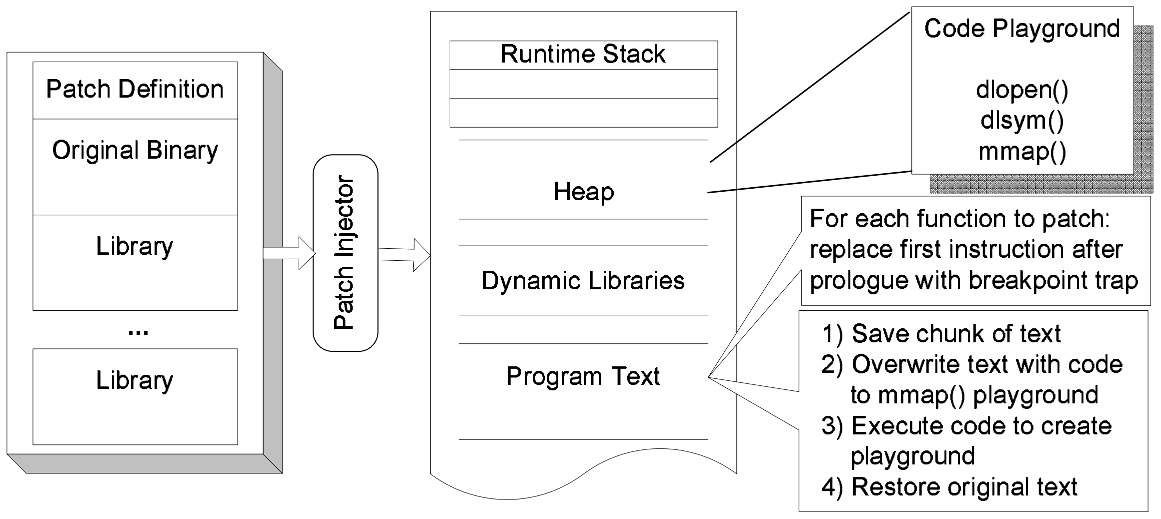
For multithreaded programs, we use the thread debugging library to obtain the necessary information for all the treads. In particular, we need to obtain the address and length of the function body which would terminate the stack inspection algorithm. Another requirement is that all threads are stopped during the setup as we need to modify the code segment shared among all the threads in order to create the code playground and insert breakpoints.

The code playground is a page within the child's memory address which gives us a predictable place to execute code on the behalf of the patched thread. The playground is created by temporarily inserting code that calls `mmap(2)` into the child's text segment and removing it when we are done. The purpose of the code playground is to make calls to `dlopen(3)` and `dlsym(3)` to load the new versions of the code into the child thread.

The indirection table is another crucial segment in the child's memory space and is required for patch application. This table stores the starting addresses of the new functions. The addresses are used in the indirect jump which we place in the old version of the code.

Before the execution of the child thread(s) is resumed, we need to be notified by the child so we can make progress with the patch. In particular, we want to be notified when the child thread enters one of the functions to

Figure 3 Patch injection overview



be patched (as this makes the thread's stack unsafe), and we want to know when the stack unwinds and the thread returns from executing the old version of one of the functions to be patched. The latter is performed by the stack inspection mechanism and is described in a subsequent section. The former is performed by placing the breakpoint instruction at the beginning of every function we are changing. Race conditions are not an issue at this point as all of the threads are still stopped.

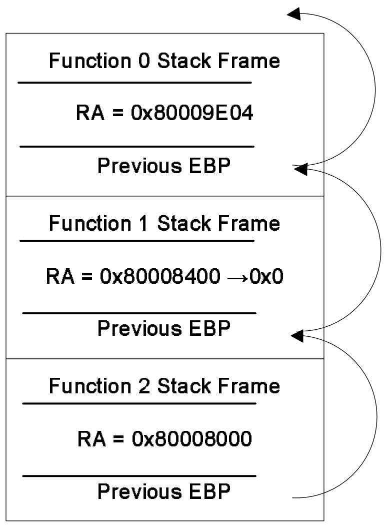
The setup ends with the insertion of break points at the first instruction of every function that needs to be patched. Once the setup phase is complete, we resume the thread(s) and wait for a signal indicating either a call to or return from one of the old functions.

5.2.2 Stack inspection

The desired invariant is that an old version of a patched function should not be calling a new version of a function that is also being patched. To make sure that the invariant holds, the stack inspection must ensure that all functions that need to be changed are not on the stack at the time when the patch is applied. We first describe the procedure for stack inspection for a single-threaded program and later extend it to the multithreaded case.

The stack is unwound all the way up to the function where execution of the program commenced, i.e., `main`. The frame pointer is used to obtain the previous frame pointer and the return address — the process depicted in Figure 4. If some return address takes us back within a patched function, one of the function already on the stack is actually being patched. The stack is considered safe if we are able to walk all the way up to `main` without detecting any of the patched functions.

Figure 4 Stack inspection and rewriting



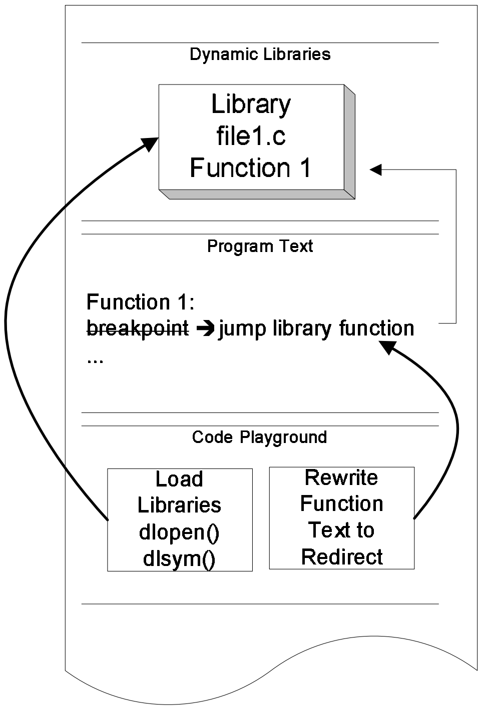
If the return address does indeed lie within the bounds of one of the functions that we are attempting to patch, the top most such function on the stack is found and its return address is replaced with the `NULL`, causing the patched process to issue a `SIGSEGV` signal when that function returns. The patch injector is awoken by the `SIGSEGV`, at which point the patch can be applied safely and the program can be restarted at the original return address.

For a multithreaded program, we need to ensure that *all* of the threads are not running any of the old versions of the code when we apply a patch. One way that this can be done is by inspecting all the threads' stack and deciding whether applying a patch is safe if every thread's stack is safe, but this incurs considerable latency overhead. Instead, we maintain a list of threads that are not in a safe state (based on the single threaded criterion of safety) and apply the patch when this list becomes empty. The list is initialized when the patch injector first attaches to the process and stops all of the threads in order to gather the necessary information about them. This is the only point where every thread's stack is inspected at the same time.

A thread is added to this list when it hits a breakpoint on entering an old version of a patched function. A thread is removed from this list when the notification of its return from the top most patched function on the stack arrives. At this point we can observe the size of the list, and proceed with patching if the list is empty.

5.2.3 Patching process

Figure 5 Function redirection


Dynamic Libraries
Library file1.c Function 1
Program Text
Function 1:
~~breakpoint~~ ➔jump library function
...
Code Playground
Load Libraries dlopen() dlsym()
Rewrite Function Text to Redirect

Once it is determined through the stack inspection mechanism that a patch can be applied, the patch process is carried out atomically as shown in Figure 5). The patching process involves two steps: (1) loading the new version of the code into the target thread's address space and (2) overwriting the first instruction within the old code by an indirect jump to the new code. In order to load the new code we execute `dlopen(3)` and `dlsym(3)` on the behalf of the patched thread. More precisely, we force the thread to execute the two functions by placing pre-constructed code that calls `dlopen(3)` and `dlsym(3)` with the correct arguments followed by a breakpoint into the playground we have pre-allocated specifically for this purpose.

When `dlopen(3)` and `dlsym(3)` return successfully, they return with the address of the new version of a patched function. We redirect execution to the new code via an indirect jump, which involves specifying a location that stores the address of the new function in the indirection table. The atomicity of the patch is guaranteed by the fact that all the functions are patched at the same time. A race condition may arise at this point as redirects are inserted only for the single stopped thread, while some other thread might start executing the old version at the same time. This condition is made impossible since the breakpoints at the beginning of each old version act as a barrier synchronization primitive — any thread attempting to run the old version will get trapped and will be resumed in the new version.

6 Experience and evaluation

The goal of our evaluation is to asses the applicability of OPUS in a real world setting and in the context of patching real security vulnerabilities. First, we isolate the raw performance penalty imposed by OPUS in applying patches dynamically from application-specific overhead. Next, we report on the prevalence of OPUS-amenable patches in an extensive survey conducted over all CERT vulnerabilities [1] issued between December 2001 and February 2005. Finally, we present three detailed case studies of using OPUS on typical vulnerabilities and the experience acquired in the process.

6.1 Performance

In this section, we consider the raw performance penalty imposed by dynamically applying a patch. OPUS adds two different sources of overhead that can potentially affect performance: a one time cost to apply the patch and a recurring cost for each call to a patched function.

Patch application proceeds in three phases described in section 5.2: initialization of the code playground, assessment of patch application safety, and patch application. We measured the overhead of initialization and patch application phases with patches of different sizes. The overhead of initializing and applying a patch was very small, ranging from 39.25ms to 81.44ms. Additional measurements showed that this cost scaled linearly with the number of functions contained in a patch.

We did not determine the cost of detecting patch application safety since this cost is highly dependent on the state of the process at patch application time. Note, however, that after an initial assessment of patch application safety, the patch injection tool stops and inspects the process only when a specific condition with safety implications occurs (e.g., when the stack is being unwound). Otherwise, the process's performance is unaffected.

In order to measure the recurring run-time overhead introduced by our method of function indirection, we evaluated the cycle times for a series of simple functions. To arrive at a lower bound for the overhead, we tested a function that takes no arguments and does no work. Then we used a function with a simple loop to simulate a longer running function to show that the overhead is fixed regardless of the function's execution time. Finally, to show that the overhead is insensitive to the number of arguments passed to the function, we tested functions with varying numbers of arguments.

The standard deviation of the measurements is 4 cycles and the measured average ranges from 3 to 9 cycles. It is not unreasonable to conclude that the recurring overhead incurred by function indirection is fixed in all tested cases.

6.2 CERT survey

To evaluate and refine the patch model described in section 3, we examined the last several years of public application vulnerability reports available on the CERT website [1]. The goal of the survey was to determine the characteristics of the most common vulnerabilities and their associated patches, and to determine if OPUS was suitable for applying them.

Reviewing the source code for the vulnerabilities and their respective patches led us to conclude that the majority of security patches were indeed small and isolated to function bodies, a conclusion similar to that of [26]. Additionally, we identified five different classes of vulnerabilities that were most prevalent and proved to be small and isolated in practice: (1) buffer overflows, (2) failed input checks, (3) format string errors, (4) logic and off-by-one errors, and (5) memory errors (double frees and leaks).

Our survey of the CERT announcements proceeded in three phases. First, we performed a high level classification on every CERT announcement issued between December 2001 and February 2005. We examined the text of each vulnerability description to determine whether the vulnerability could potentially be amenable to patching with OPUS. We looked for two things in this first high level pass: (1) that the vulnerability affects an application written in C for a commodity operating system and (2) that the description unambiguously placed the vulnerability into one of the five most common error categories mentioned above. In the second stage of our survey, we attempted to locate and inspect the source code for as many of the suitable vulnerabilities as possible. Unfortunately, since many of the vulnerabilities that made our first cut affected closed source systems, we were limited in the actual number we could actually inspect. Of the vulnerabilities we were able to inspect by hand, we further attempted to find the original patch associated with the vulnerability that was released by the vendor. This helped us to understand the characteristics of real-world vendor patches, and provided us with actual fixes for our third phase of the survey. In the final phase, we took several real-world patches from vendors and ran them through OPUS to validate the intuition behind our patch model.

Table 1 gives a summary of our survey results. Of the 883 CERT notifications we examined, 445 (50.4%) were found to be amenable to patching with OPUS based on the description of the vulnerability. Since gauging the amenability of patches by reading the description is hardly conclusive, we examined the source code of 115 out of the 445 CERTs and found, through inspection, that given the constraints of our patch model, 111 were amenable. Finally, of the 111 CERT vulnerabilities we examined by hand, we ran 26 real patches through OPUS to verify that the patch was dynamically applicable and to ensure that it did in fact close the vulnerability without adversely affecting the application. Of the 26 patches we tested 22 were successfully applied with OPUS. The four failures were due to implementation bugs in our current prototype, which we are in the process of fixing.

6.3 Experience with OPUS

To evaluate the practicality of using OPUS with real world security patches, we tested OPUS on 26 vulnerabilities taken from our CERT survey. We took each vulnerability through OPUS from start to finish: static analysis, patch generation, patch injection, and patch verification. In many cases, patches were taken from start to finish without intervention on our part. In cases where the patch was part of a larger upgrade, we had to manually isolate the changes relevant to the individual vulnerability. The reason is that the OPUS patch generation component cannot automatically distinguish between patches and feature upgrades.

Table 2 gives a summary of the vulnerabilities tested. In the table, *Patch Type* represents whether or not the patch used in testing was directly from a vendor patch (Source) or distilled from a version upgrade (Upgrade). The *Testing Type* represents one or more methods we used to determine the success of the patch:

- **Exploit**: The client was tested with an exploit program associated with the vulnerability
- **Load**: The client was placed under a heavy load during and after patching

Amenable Vulnerabilities	
Amenable by description (no source)	334
Amenable by inspection (source)	111
Amenable by application (patch applied)	22
Non-amenable Vulnerabilities	
Not amenable by description	212
Not amenable by inspection	4
Not amenable by application (failed patch)	4
Not written in C	56
Router or embedded OS	90
Other	78
Total	883

Table 1: Survey of CERT Vulnerabilities and Corresponding Patches

- **Codepath**: The code path through the replaced functions was exercised
- **Operational**: The operation of the program was checked after patching by normal interaction

Testing OPUS on each of the 26 vulnerabilities was an arduous process that spanned several weeks of concentrated work. Specifically, for each application, testing involved locating the source code and corresponding patch, getting the application code to compile on a modern version of Linux (Fedora Core 2), separating patches from feature upgrades, finding and applying exploits on patched applications, and in the cases where exploits weren't available, devising them ourselves. Many of these steps required a good understanding of the source code. Nevertheless, we expect the process to be significantly more streamlined for the developers of these applications.

6.4 Case study: patching real vulnerabilities

In order to highlight our experience of patching real vulnerabilities with OPUS, we provide case studies of three different patches: one for the Apache webserver and two for the MySQL DBMS.

6.4.1 Apache chunked transfer bug

For our first evaluation, we selected a vulnerability in the Apache webserver's handling of chunked transfer encoding [2, 8]. This particular vulnerability was the source of the Scalper exploit [20] and was the precursor to the Slapper worm [4]. The vulnerability itself is a simple failure to properly handle negative chunked encoding sizes, which leads to a buffer overflow that can cause execution of arbitrary code. We obtained a version of the exploit attack code that was available on the web and used it to test the success of the patch [13, 9].

The patch affected 7 functions in one file (including one new function) and consisted of 16 changed lines of source and 37 new lines. We were able to successfully take the patch from source all the way through to patching a running Apache process.

Functional evaluation. Ultimately, we used the exploit attack code to make sure the patch correctly fixed the vulnerability. Running the attack code on the unpatched process caused segmentation faults in the forked request handlers, while running the attack code after the patch was applied resulted in a nicely formatted error message returned to the attack client.

Front end experience. Because the patch was relatively complex, it helped to expose several bugs in our front end processing and helped to refine our annotation format. The source file that was patched contained static functions that did not change, which initially caused our patch generation tool to break. After examining our handling of static functions with the patch, we were able to make several implementation fixes and clarify our understanding of how static functions should be handled by OPUS.

Back end experience. We performed the online patch on a running copy of Apache 1.3.24 under a simulated heavy load using SPECweb99. We tested patch injection on both the *forked worker* and *thread pool* modes (using 200 threads), showing that OPUS works well on real single threaded and multithreaded applications.

Conclusion. We were able to show success fixing a wild exploit using a patch that was developed without any foresight of using OPUS, which meets the goals set

Application	*CERT ID*	*Vulnerability Type*	*Patch Type*	*Result*	*Testing Type*
Apache 1.3.24	944335	Buffer overflow	Upgrade	Pass	Load/Exploit
Apache 2.0.50	481998	Buffer overflow	Upgrade	Pass	Load/Codepath
BIND 4.9.5	13145	Buffer overflow	Source	Pass	Exploit
BIND 4.9.5	868916	Input checking	Source	Fail	None
BIND 4.9.6	CA-1997-22	Input checking	Upgrade	Pass	Exploit
BIND 4.9.7	572183	Buffer overflow	Source	Pass	Operational
BIND 8.2	CA-1999-14	Multiple bugs (4)	Source	Pass	Exploit
BIND 8.2.2	16532	Buffer overflow	Source	Pass	Exploit
BIND 8.2.2	196945	Buffer overflow	Source	Pass	Exploit
BIND 8.2.2	325431	Input checking	Source	Pass	Exploit
BIND 8.2.2-P6	715973	DoS	Source	Pass	Exploit
BIND 8.2.2-P6	198355	DoS	Source	Pass	Exploit
BIND 9.2.1	739123	Buffer overflow	Source	Pass	Exploit
freeRadius 1.0.0	541574	DoS	Upgrade	Pass	Operational
Kerberos 1.3.4	350792	Double free	Upgrade	Fail	None
mod_dav 1.91	849993	Format string	Source	Pass	Codepath
MPlayer 0.91	723910	Buffer overflow	Upgrade	Pass	Operational
MySQL 4.0.15	516492	Buffer overflow	Upgrade	Pass	Exploit
MySQL 4.1.2	184030	Input checking	Upgrade	Pass	Exploit
rsync 2.5.5	325603	Buffer overflow	Upgrade	Fail	None
Samba 2.2.6	958321	Buffer overflow	Upgrade	Pass	Exploit
Samba 3.0.7	457622	Buffer overflow	Upgrade	Pass	Codepath
Samba 3.0.9	226184	Buffer overflow	Source	Pass	Codepath
Sendmail 8.12	398025	Buffer overflow	Source	Pass	Operational
Squid 2.4	613459	Buffer overflow	Upgrade	Fail	None
tcpdump 3.8.1	240790	Buffer overflow	Upgrade	Pass	Operational

Table 2: Summary of real patches tested with OPUS

forth in developing OPUS.

6.4.2 MySQL password bugs

For our second application experience, we evaluated MySQL–a service which is more stateful than Apache, and thus would have a higher cost to shutdown and patch. We chose two simple vulnerabilities found in the MySQL database management application. The first allows a local or remote user to bypass authentication with a zero-length password [30]. The second exploits a buffer overrun in the password field allowing execution of arbitrary code [31]. We obtained exploits available on the web for both to help evaluate the success of the patch [24, 19].

Patch characteristics. Both patches supplied by the vendor for these vulnerabilities were very simple "one-liners" that changed either a single line or a handful of lines contained within a single function. From our survey of common vulnerabilities and patches, this is a very common characteristic of buffer overflow patches.

Functional evaluation. We were able to successfully patch the running MySQL service while it was under a simulated load from a simple database performance benchmark (sql-bench). Running the first attack on the unpatched process allowed us to gain access to the DBMS server and running the second attack allowed us to crash the server. After applying the patch, both exploits failed to compromise the server, and both were returned a proper error message.

Front end and back end experience. Because of the rather simple nature of the patches themselves and the fact that we performed this test after our experience with Apache, these particular patches did not uncover any new issues in either the front or back end. The patches were both successfully generated and applied with little difficulty.

Conclusion. The MySQL case study is significant in that it shows OPUS can work with more stateful applications (e.g., database servers). These stateful applications are most likely to benefit from avoiding the restart associated with the application of traditional patches. Moreover, stateful services such as database servers offer a high opportunity cost for those seeking to exploit vulnerabilities. Thus, the ability to successfully patch services like MySQL is an important validation of our work.

6.5 Utility of static analysis

Was our static analysis useful in producing safer dynamic security patches? The short answer is no. The reason is that only a hand-full of the security patches we examined modified non-local program state. For the few patches that did modify non-local program state, we used our understanding of program semantics to determine that the corresponding modifications were in fact not dangerous. As an example, consider the following excerpt from BIND 8.2's "nxt bug" patch [28]:

```
...
if ((*cp & 0x01) == 0) {
  /*
   * Bit zero is not set; this is an
   * ordinary NXT record.  The bitmap
   * must be at least 4 octets because
   * the NXT bit should be set. It
   * should be less than or equal to 16
   * octets because this NXT format is
   * only defined for types < 128.
   */
   if (n2 < 4 || n2 > 16) {
      hp->rcode = FORMERR;
      return (-1);
   }
}
...
```

The above code checks if a field in the incoming message's header is properly formed, and if it is not, it writes an error code (`FORMERR`) to a memory location on the heap (`hp->rcode`) and returns -1 to indicate failure. We know that the write is benign: upon return from the patched function, BIND checks the value of `hp->rcode` for the error type and outputs a corresponding error message. However, the OPUS static analysis issues the following false warning: `error: 2089: writing to dereferenced tainted pointer (hp)`. We encountered similar warnings in our evaluations, but our understanding of the source code allowed us to quickly discard them as false positives.

7 Related work

Dynamic updates: Many existing works in dynamic software updating make use of strong programming language support (e.g., dynamic binding and type-safety as provided in Lisp, Smalltalk, and Java) [10] [16]. All of these approaches target a wide class of software updates—not just security patches—and can make strong guarantees about the safety of a runtime patch. OPUS, in contrast, does not assume strong language support nor can it perform arbitrary upgrades. In fact, a fundamental design criteria of our system is that it must be able to handle existing, widely-deployed software and consequently, our decision to target C applications reflects this generality vs. practicality tradeoff.

Dynamic update techniques that don't rely on strong language support have also been explored. Early work by Gupta *et al.* [14], for example, targets C applications and is the closest to ours in technique, but they neither target security patches nor do they use static analysis to estimate patch safety. More recently, Stoyle *et al.* [26] presented a dynamic updating system for a C-like language that provides strong safety guarantees. Although more general in the type of patches it admits, their system requires software updates to be written in a special-purpose language; true support of C is cited as future work. While OPUS does not provide strong safety guarantees, it does not require that applications be constructed in a custom language.

Shield: Shield [32] is a system of vulnerability-specific network filters that examine incoming and outgoing network traffic of vulnerable applications and correct malicious traffic en-route. Properly constructed Shield policies can be more reliable than conventional patches and like dynamic patches, applying policies in end hosts is a non-disruptive procedure. To distinguish our work from Shield, we note the following differences:

- Shield requires the programmer to specify all vulnerability approach vectors—a task that involves significant programmer effort and risks introducing false positives as well as false negatives when dealing with complicated applications. Unlike Shield, OPUS does not require the programmer to specify a vulnerability state machine. Little programming effort beyond what would be required to construct a conventional patch is necessary.
- While Shield can defend against network-borne pathogens quite effectively, it cannot defend against file-system worms, protocol-independent application vulnerabilities (e.g., bugs in a script interpreter), or memory allocation problems not tied with any specific malicious traffic. In contrast, OPUS can defend against most vulnerabilities that can be fixed via conventional security patching.
- Monitoring network traffic on a per-application basis induces a performance penalty on Shielded ap-

plication that is proportional to the amount of network traffic. Dynamic patches result in negligible performance overhead once applied.

Redundant hardware: Redundant hardware offers a simple, high-availability patching solution. Visa, for example, upgrades its 50 million line transaction processing system by selectively taking machines down and upgrading them using the on-line computers as a temporary storage location for relevant program state [22]. However, Visa's upgrade strategy is expensive and as a result precludes use by those with fewer resources. Perhaps more severe, it requires that developers and system administrators engineer application specific upgrade strategies, thereby adding to the complexity of development and online-evolution [16]. Our standpoint is that ensuring system security should neither be expensive nor require ad-hoc, application-specific solutions.

Microreboots: Microreboots [7] provide a convenient way to patch applications composed of distinct, fault-tolerant components—install the new component and then restart it. While a microreboot approach to patching may be viable for enterprise web applications, it cannot serve as a generic non-disruptive patching mechanism. The reason for this is that a microrebootable system must be composed of a set of small, loosely-coupled components, each maintaining a minimal amount of state. OPUS differs from microreboots in that it makes no assumptions about the coupling of software components: a monolithic system can be patched just as easily as a heavily compartmentalized system.

8 Future work

8.1 Prototype

In order to perform stack inspection, our current prototype performs a backtrace on the stack using frame pointers and return addresses. Some functions, however, are compiled to omit frame pointers (e.g., several functions in GNU libc). Furthermore, stack randomization tools make it difficult to determine the structure of the stack. While we have a makeshift solution to deal with these problems, it insists that applications preload wrapper libraries–a requirement that somewhat tars our goal of "no foresight required". Thus, we are currently exploring more transparent mechanisms to deal with these issues.

Many security patches are targeted at shared libraries. While the current implementation of OPUS cannot dynamically patch libraries, the ability to do so would be valuable in closing a vulnerability shared by several applications. Thus, we are working on extending our `ptrace`-based stack-inspection mechanism to work with multiple processes, all of whom share a common vulnerable library.

Finally, many system administrators choose to turn off `ptrace` support, leaving OPUS unable to function. To deal with this issue, we are currently working on hardening `ptrace` support for OPUS.

8.2 Static analysis

Assessing the safety of a dynamic patch is undecideable in the general case, so the burden falls on the static analysis to alert the user of all possible changes that may fault the application when a patch is applied. With respect to tracking writes to new non-local data, the current implementation of static analysis could use a tighter bound on the taint set. This can be accomplished by implementing proper support for multi-level pointer variables (one can think of structs and multi-dimensional arrays as multi-level pointer variables). A more sophisticated algorithm to compute pointer aliases and associated taintings is also being considered. The analysis could also benefit from better handling of explicit casts and non-straightforward uses of the C type system.

In addition to the above refinements, we are considering implementing path-sensitive taint flow analysis which would effectively re-enable warnings for all blocks (as if they were new to the patch) depending on some variable being assigned a new value in the patched code.

Finally, the success of static analysis hinges on our ability to tell which program fragments are new. Currently, this is accomplished by diff-ing the source trees, a method that is too imprecise to arrive at a complete set of statements being modified if the correspondence between statements and line numbers is anything but uniform. We are currently considering program differencing [17] as an alternative to shallow diffs.

9 Conclusion

Despite our attempt to alleviate safety concerns through static analysis, the complexity introduced by dynamic update, although often negligible when applied to security patches, makes the hard problem of ensuring patch reliability even harder. In the end, the added complexity may deter developers from adopting the technology or worse, prevent users from patching their systems more quickly. However, by looking at a large sample of real security vulnerabilities, we have shown that a significant number of applications within our problem scope could have been safely patched with OPUS had OPUS been available at the time of vulnerability announcement. This result strongly supports our claim that dynamic security patching is safe and useful in practice. To this effect, we have presented a viable alternative to the traditional security patching methodology.

10 Acknowledgments

We thank the anonymous reviewers, Nikita Borisov, Eric Brewer, our shepherd Peter Chen, David Wagner, and the Berkeley SysLunch and Security reading groups for their valuable feedback.

References

[1] US-CERT Vulnerability Notes Database. `http://www.kb.cert.org/vuls/`.

[2] Apache security bulletin. `http://httpd.apache.org/info/security_bulletin_20020617.txt`, June 2002.

[3] ARBAUGH, W. A., FITHEN, W. L., AND MCHUGH, J. Windows of vulnerability: A case study analysis. In *IEEE Computer* (2000).

[4] ARCE, I., AND LEVY, E. An analysis of the slapper worm. *IEEE Security and Privacy 1*, 1 (2003), 82–87.

[5] BEATTIE, S., ARNOLD, S., COWAN, C., WAGLE, P., WRIGHT, C., AND SHOSTACK, A. Timing the application of security patches for optimal uptime. In *LISA* (2002), USENIX, pp. 233–242.

[6] BREWER, E. Lessons from giant-scale services. In *IEEE Internet Computing* (Aug. 2001).

[7] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot - a technique for cheap recovery. In *Proceedings of the 6th Operating System Design and Implementation* (Dec. 2004), pp. 31–44.

[8] CERT. CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. `http://www.cert.org/advisories/CA-2002-17.html`, June 2002.

[9] DTORS.NET. Apache chunked encoding example exploit. `http://packetstormsecurity.org/0207-exploits/apache-chunk.c`.

[10] DUGGAN, D. Type-based hot swapping of running modules (extended abstract). In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (2001), ACM Press, pp. 62–73.

[11] DUNAGAN, J., ROUSSEV, R., DANIELS, B., JOHNSON, A., VERBOWSKI, C., AND WANG, Y.-M. Towards a self-managing software patching process using persistent-state manifests. In *International Conference on Autonomic Computing (ICAC) 2004* (2004).

[12] FREE SOFTWARE FOUNDATION, INC. *Using the GNU Compiler Collection*. Boston, MA, USA, 2004.

[13] GOBBLES SECURITY. Apache "scalp" exploit. `http://www.hackindex.org/boletin/0602/apache-scalp.c`.

[14] GUPTA, D., AND JALOTE, P. On-line software version change using state transfer between processes. *Softw., Pract. Exper. 23*, 9 (1993), 949–964.

[15] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng. 22*, 2 (1996), 120–131.

[16] HICKS, M., MOORE, J. T., AND NETTLES, S. Dynamic software updating. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), ACM Press, pp. 13–23.

[17] HORWITZ, S. Identifying the semantic and textual differences between two version of a program. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation* (June 1990), pp. 234–245.

[18] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990.

[19] K-OTIK SECURITY. Remote MySQL Priviledges Exploit. `http://www.k-otik.com/exploits/09.14.mysql.c.php`.

[20] MITUZAS, D. Freebsd scalper worm. `http://dammit.lt/apache-worm/`.

[21] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformations of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 213–228.

[22] PESCOBITZ, D. Monsters in a box. *Wired 8*, 12 (2000), 341–347.

[23] RESCORLA, E. Security holes . . . who cares? In *12th Usenix Security Symposium* (Washington, D.C., August 2003), pp. 75–90.

[24] SECURITEAM.COM. Local and Remote Exploit for MySQL (password scrambling). `http://www.securiteam.com/exploits/5OP0G2A8UG.html`.

[25] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium* (Washington, D.C., Aug. 2001).

[26] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, L. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), pp. 183–194.

[27] THE SOFTWARE DEVELOPMENT LIFE CYCLE TASK FORCE, S. A. Improving security across the software development lifecycle. Tech. rep., National Cyber Security Partnership, April 2004.

[28] US-CERT. Vulnerability Note VU#16532 BIND T_NXT record processing may cause buffer overflow. `http://www.kb.cert.org/vuls/id/16532`, November 1999.

[29] US-CERT. Vulnerability Note VU#715973 ISC BIND 8.2.2-P6 vulnerable to DoS via compressed zone transfer, aka the zxfr bug. `http://www.kb.cert.org/vuls/id/715973`, November 2000.

[30] US-CERT. Vulnerability Note VU#184030 MySQL fails to properly evaluate zero-length strings in the check_scramble_323() function. `http://www.kb.cert.org/vuls/id/184030`, July 2004.

[31] US-CERT. Vulnerability Note VU#516492 MySQL fails to validate length of password field. `http://www.kb.cert.org/vuls/id/516492`, September 2004.

[32] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of SIGCOMM '04* (Aug. 2004).