Efficient Online Validation With Delta Execution

Joseph Tucek   Weiwei Xiong   Yuanyuan Zhou

Department of Computer Science, University of Illinois at Urbana-Champaign

tucek@illinois.edu   wxiong2@illinois.edu   yyzhou@illinois.edu

Abstract

Software systems are constantly changing. Patches to fix bugs and patches to add features are all too common. Every change risks breaking a previously working system. Hence administrators loathe change, and are willing to delay even critical security patches until after fully validating their correctness. Compared to off-line validation, on-line validation has clear advantages since it tests against real life workloads. Yet unfortunately it imposes restrictive overheads as it requires running the old and new versions side-by-side. Moreover, due to spurious differences (e.g. event timing, random number generation, and thread interleavings), it is difficult to compare the two for validation.

To allow more effective on-line patch validation, we propose a new mechanism, called delta execution, that is based on the observation that most patches are small. Δ execution merges the two side-by-side executions for most of the time and splits only when necessary, such as when they access different data or execute different code. This allows us to perform on-line validation not only with lower overhead but also with greatly reduced spurious differences, allowing us to effectively validate changes.

We first validate the feasibility of our idea by studying the characteristics of **240** patches from 4 server programs; our examination shows that 77% of the changes should not be expected to cause large changes and are thereby feasible for Δ execution. We then implemented Δ execution using dynamic instrumentation. Using real world patches from 7 server applications and 3 other programs, we compared our implementation of Δ execution against a traditional side-by-side on-line validation. Δ execution outperformed traditional validation by up to 128%; further, for 3 of the changes, spurious differences caused the traditional validation to fail completely while Δ execution succeeded. This demonstrates that Δ execution can allow administrators to use on-line validation to confidently ensure the correctness of the changes they apply.

Categories and Subject Descriptors D.4.5 [*OPERATING SYSTEMS*]: Reliability; D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification

General Terms Reliability, Verification

Keywords Delta Execution, Patch Validation, Testing

1. Introduction

Modern software systems are plagued by failures. As software has grown in size and complexity, the difficulty of finding and fixing bugs has increased. Inevitably, these bugs leak into production run software, contributing up to 26-30% of system failures (Marcus and Stern 2000), and costing the US economy upwards of $59 billion yearly (National Institute of Standards and Technlogy (NIST), Department of Commerce. 2002). Our increasing reliance on computers for everything we do implies that reliable software is becoming ever more important.

Unlike a bridge or a building, however, software systems don't suffer from physical wear; bits don't rot. Unless something changes, we expect a working software system to continue to work. Unfortunately, changes are all too common. Whether to repair flaws or add features, software patches are released all the time. For instance, Microsoft releases patches monthly on "Patch Tuesday", the second Tuesday of the month (Microsoft 2003). This is twelve times a year that the software is changed, potentially causing working systems to fail. Other software fares no better; in 2007 over 7600 vulnerabilities were reported to CERT (CERT); that is 7600 changes just to fix security holes. Our software systems are far from being static.

These changes are a common and continuing source of failures. For instance, Microsoft recently withdrew SP1 for Windows Vista for causing "computers to crash or enter an endless cycle of boots and reboots" (McDougall 2008). Similarly, Apple's recent OS X update irritated users by breaking applications (Pegoraro 2008). Nearly 70% of patches are buggy in their first release (Cowan et al. 2000; Rescorla 2003; Sidiroglou et al. 2007); clearly patching is a risky business. Hence, administrators loathe to be the first to apply a

```
Patch: tar null pt
tar/src/incremen.c
....
+if(dirp){
   if(children!=NO_CHILDREN)
     for(entry=dirp; ...){
       //main loop
     }
   free(dirp);
+}
```

(a) GNU tar patch

```
Patch: CAN-2004-0493
httpd-2.0/server/protocol.c
....
+if((fold_len-1)>
+    r->server->limit){
+  r->status = BAD_REQUEST;
+  return;
+}
```

(b) Apache Patch

```
Patch: CAN-2004-0811
httpd-2.0/server/core.c
....
 if(new->satisfy[i] !=
    SATISFY_NOSPEC){
   conf->satisfy[i]=new->satisfy[i];
+}else{
+  conf->satisfy[i]=base->satisfy[i];
+}
```

(c) Apache Patch

Figure 1. Three patches fixing real bugs. Lines with a '+' indicate inserted code. Slightly simplified for descriptive purposes.

patch, and even for critical security patches, they will delay to allow time to validate patch correctness (Beattie et al. 2002). Indeed, Patch Tuesday was created to give administrators a predictable schedule for validation and roll out (Microsoft 2003). Yet these delays run counter to minimizing the window of vulnerability (Browne et al. 2001; Joshi et al. 2005). Administrators have two bad options: patch now and risk a bad patch, or patch later and risk being exploited. Currently, the balance favors waiting (Beattie et al. 2002); patching without validating is too risky.

1.1 Validating Patches

In general, there are two means of validating a patch: off-line (Barrett et al. 2004) and on-line (Cook and Dage 1999). Off-line validation is appealing due to its simplicity–merely set up a test system and see if it works. Yet this has the same shortcoming as the vendor's testing does: the test environment may not reflect actual production usage. Since the vendor's tests clearly don't filter out buggy patches, off-line validation seems unlikely to solve our troubles. On-line validation, with a live workload, is preferred, as it is more accurate (Nagaraja et al. 2004). In on-line validation, an instance of the "new" system is run simultaneously with the unmodified production run instance. Both are fed the real production workload, and the results are compared. This exercises the whole system, covers extreme or unusual site-specific conditions, and allows an administrator to answer the only important question: "*does it break **my** workload*". On-line validation's realism provides a high level of assurance.

Unfortunately, there are difficulties to on-line validation. Since two full instances (production and testing) must be run, we will use additional computational resources. These instances may be run on separate hardware, or isolated on the same hardware (e.g. through a VMM (Lowell et al. 2004)). Running on the same hardware can hurt performance. Indeed, competition between the original instance, the test instance, and the validation routines reduced the performance of the production service by up to 2.6x in our experiments.

Validating with a duplicated hardware setup is equally unsatisfactory. Since the test instance must support the same load as the production run instance, a separate validation setup is expensive. Although the extra hardware may be affordable, other costs (e.g. power, cooling, floorspace, software & support licenses, etc.) dominate, especially the cost of administrator labor. Configuring a system which is identical to the production instance (but different in small ways so as to not adversely affect the production side) represents an expensive drain of human labor.

Even if money and performance don't matter, worse yet is that actually verifying that the output of the test instance is correct is non-trivial (Nagaraja et al. 2004). Non-determinism due to thread interleaving, randomization, and small timing differences can cause differences in the output which frustrate checking the correctness of the test instance. Even for identical side-by-side copies, these spurious differences can be quite large, and even make validation completely impossible. Hence, although on-line validation gives strong evidence that a patch won't break when put into production, given the options of expense vs. overhead and trouble with spurious differences, it is not surprising that on-line validation is uncommon.

1.2 Multiple Almost Redundant Executions

The large challenges of on-line validation seem disproportionate compared to the size of the changes. Figure 1.a shows a patch for a real bug in tar. The only difference is that the main processing loop is made conditional on `dirp` being non-null. The patch is only two lines long; the omitted main loop is 100 lines long. Also, in practice `dirp` will almost always be non-null and the patch will have no effect. We would expect the execution of the patch and the original to be identical, down to the individual instructions, almost all of the time.

We call such highly similar executions MAREs: multiple almost redundant executions. That is, we have more than one execution that are almost, but not quite, completely identical and redundant to each other. Such cases of MAREs are common in patch validation, especially for security patches. Figure 1.b and 1.c show two security patches for the Apache web server. Both are small, and indicate that some action should be taken in case of a certain condition. Again, most of the time the condition will not occur, since most requests are not exploit attempts. Consider that adding a bounds check is the straightforward way to deal with buffer overflow. The patch is small, and in the case where there is no exploit attempt, nothing at all happens. Together, these imply that security patches will only ever lead to small changes in execution. Overall, it seems likely that validating the correctness of patches will be rife with uselessly redundant work.

1.3 Contributions

To eliminate redundancy in these MARE tasks, we explore a new technique called *delta (Δ) execution*. Δ execution (illustrated in Figure 2) runs only the differences, or deltas, of the two different versions separately. Mostly, only one execution is necessary (such runtime is called *merged execution*). In the few short segments where the original and the patched versions differ(called Δ code), we run two separate executions (called *split execution*). After a time, we logically merge the two separate executions back into one (making note of state differences, or Δ state), and continue running merged. Further executions of the patch, or accesses to data which differs, causes another split, with each half running as if it had been running alone the entire time.

For patch validation, Δ execution should have several key advantages:

Lower overhead when merged. We expect that the bulk of execution between the two versions will be identical, and most runtime will be merged. Most computation, I/O, and system calls will be done once. There will be little competition between instances for resources. Especially if a user's particular workload never triggers the change, the overhead will be much lower than running two separate instances.

Lower overhead when split. Even when split, resource competition is minimized. Rather than issue I/O operations and system calls twice, the Δ execution runtime can monitor expensive operations and ensure that those which are identical in both instances are issued only once, with the two instances transparently sharing the result. This especially limits contention for disk resources.

Easier validation due to more similar executions. Δ execution eases validation by reducing non-determinism. Mostly the two instances will run merged, and any sources of non-determinism (e.g. thread interleaving) will influence both versions identically. Only non-determinism during the short segments of split execution will affect the output of the two versions.

These advantages allow Δ execution to support efficient and effective patch validation. To ensure the feasibility of Δ execution, we performed a study of patches. We manually looked at 60 patches each from MySQL, Apache, OpenSSL, and Squid, for a total of 240 patches. As detailed in Section 2, we find that 77% of the patches should be straightforward to run under Δ execution, and that extensions to Δ execution should not only improve the performance but allow a further 6% of patches to be supported.

Further, we demonstrate a practical implementation of Δ execution, detailed in Section 4. We test Δ execution with 10 different patches, using software ranging from servers like Apache, to multi threaded utilities like crafty. Even with the overhead of instrumentation, Δ execution outperforms basic side-by-side validation by 12%, due to competition for resources and the overhead of checking correctness. If we account for the intrinsic overhead of the instrumentation engine, this increases to 74%. Although both Δ execution and our basic side-by-side validation implementation detect incorrect runs, Δ execution was unaffected by non-determinism from the underlying programs. The basic side-by-side verification implementation was confused by non-determinism, and hence flagged many correct runs as buggy.

Overall, Δ execution is capable of efficiently dealing with the MAREs found in patch validation. The overhead is far less than running two separate instances, and it is easier to identify when a true discrepancy occurs. This greatly increases the practicality of on-line patch validation; hence, Δ execution may be an important mechanism for ensuring the reliability of future software systems.

2. Real World Patch Characteristics: Feasibility Analysis of Delta Execution

In order for delta execution to be effective, the differences between the original and modified executions must be small. The three patches previously shown in Figure 1 are all five lines or shorter. Indeed, patches are generally small and unobtrusive for good reason. Large or intrusive patches are hard to understand. Hence they are seen as being more likely to introduce bugs. Some software projects, such as the Linux kernel, discourage large patches (Torvalds 2000). If a large change must be made, the programmer is encouraged to provide it as several smaller changes, which can be applied and verified separately. Small unobtrusive changes are easier to understand, and are viewed as less likely to hide or introduce new faults. This is especially true for security patches; the goal is to fix the problem as quickly and as simply as possible. Hence, it is unsurprising that many patches are small.

To back this intuition with data, we performed a study of real world patches. We manually evaluated patches from four representative open source applications (Apache, MySQL, OpenSSL, and Squid), and categorized them according to how we believe they will behave under Δ execution. Further, we determined 10 total general categories to classify them under, based on distinguishing characteristics. We choose 60 patches from each of these large and mature applications, for a total of 240 patches.

2.1 10 categories of patches

Table 1 shows the 10 categories of patches we identified, while Figure 3 shows concrete examples, drawn from the applications we studied and used in our experiments. These categories are meant to summarize essential qualities of the patches as they relate to Δ execution and are not necessarily exhaustive. However, all of the changes we studied fall

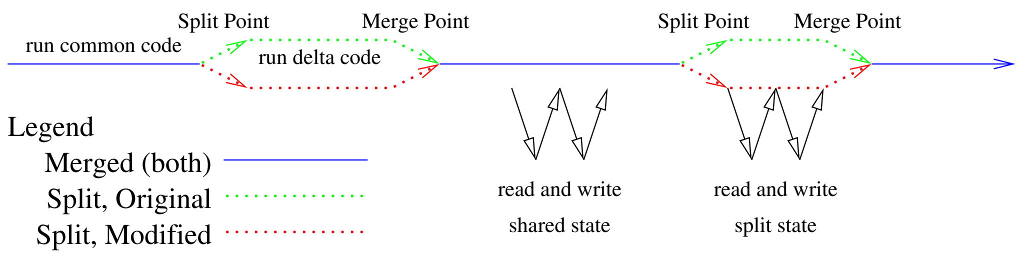


Figure 2. Delta execution "fakes" running two instances by running common segments only once, in *merged execution*. Non-common segments (due to "*delta code*" or "*delta state*") are run separately, in *split execution*.

Definitions of patch categories			
Category	**Description**	**Abbr.**	**Δ-EXE**
Refactor	Changing names of variables, functions, etc	refact	all
Rare path	The code that is changed but almost never be run	rare	all
Stack effect	The changes are expected to affect the stack	stack	most
Side effect	Changes that make global effects (i.e. heap manipulation, different I/O.)	side	some
Conditional	Adding or changing small conditions (i.e. adding a buffer overflow check)	cond	most
Synchronization	Changes that are related to concurrency (i.e. data race prevention, deadlock elimination.)	sync	most
Data structure	Changing a structure size	struct	none
Macro	Patches that involve modifying macros	macro	most W/ C
Polymorphic	Data type changes that don't actually affect memory layout	poly	most W/ C
Complex changes	Changes that are too complex to segregate to small patches (i.e. hundreds of lines affected, new files added.)	comp	none

Table 1. Patch categories. The last column of the table indicates whether patches in those categories can be well dealt with in our current Δ execution implementation or not. "W/ C" indicates YES with minimal additional compiler support.

into one or more of these categories. We also list how well we feel our current implementation of Δ execution handles each category, based on the 240 patches we examined. More specifically about each category:

Refactor "Harmless refactoring," like the renaming of a function shown in (a), or otherwise merely restructuring the organization of code, is not intended to actual change the behavior of a program. Hence, this is nearly an ideal case for Δ execution: unless the refactoring is buggy, no execution properties will change.

Rare path It isn't so much the changes in (b) that allow efficient Δ execution, but where they are: in the segfault handler. Changes to rarely used functionality, error handlers, etc. are all good candidates, since for many runs the changed code won't even be executed.

Stack effect Many patches are expected to cause little overall change outside of the their respective functions. The intended result of the function remains the same, just precisely how it is gone about differs. In (c), decrementing `count` prevents a possible overflow, and in nearly all cases has zero impact on other execution.

Side effect In contrast, some patches are not completely isolated. (d) is characterized by changing what is put into heap memory. This and similar patches will likely cause some further splitting due to differing data. The differing data does tend to stay confined (in this case to the error log buffer), but overall efficiency will be reduced.

Conditional Changes to condition statements are the most common class of patch we found. Most of them, like this Apache patch in (e), are intended to better deal with some corner case. Δ execution will work wonderfully in the common case that the result of evaluating the condition remains the same.

Synchronization Another easy case is changes in synchronization, like the addition of a lock in (f). Synchronization changes are mostly intended to remove races or deadlocks; since they have no effect unless the error occurs, and such errors are triggered only sporadically, Δ execution works very well.

Data structure Changes to data structures, such as the new field added in (g), do not work at all with our current implementation. Especially for memory-unsafe languages like C and C++, changing the size of a type can shift the entire layout of memory, which makes it difficult to identify semantically identical data. Minor help from the compiler

```
openssl/crypto/des/des_enc.c
....
- void des_encrypt(DLONG *data, k_sched ks, int enc)
+ void des_encrypt1(DLONG *data, k_sched ks, int enc){
....
```
a) Refactor

```
sql/mysqld.cc
....
  sig_handler handle_segfault(ing sig){
+   curr_time = time(NULL);
+   localtime_r(&curr_time, &tm);
....
```
b) Rare path

```
atphttpd/sockhelp.c
....
  int sock_gets(sockfd, str, count){
+   count--;
....
```
c) Stack effect

```
httpd/server/scoreboard.c
....
 if(rv!=APR_SUCCESS){
  ap_log_error(APLOG_MARK, ...,
-   "unable to create scoreboard\"%s\"",
+   "unable to create or access scoreboard\"%s\"",
....
```
d) Side effect

```
httpd/modules/proxy/mod_proxy.c
....
 if(strcmp(e[i].sch,"*")==0 ||
  (e[i].regex &&
-  regexec(e[i].rex,url,0,0,0))||
+  regexec(e[i].rex,url,0,0,0)==0)||
....
```
e) Conditional

```
openssl/crypto/rsa/rsa_eay.c
....
  int helper(RSA *rsa, BTX *ctx){
+   CRYPTO_r_lock(CRYPTO_LOCK_RSA);
    if(rsa->flags&RSA_NO_BLINDING)
      ret=1;
+   CRYPTO_r_unlock(CRYPTO_LOCK_RSA);
....
```
f) Synchronization

```
httpd/..../mod_deflate.c
....
  struct deflate_ctx_t{
    ....
+   int inflate_init;
  };
....
- if(!inflate_init++){
+ if(!ctx->inflate_init++){
....
```
g) Data structure

```
httpd/modules/cache/mod_mem_cache.c
....
- #define DEFAULT_MIN_CACHE_OBJECT_SIZE 0
+ #define DEFAULT_MIN_CACHE_OBJECT_SIZE 1
....
```
h) Macro

```
sql/log_event.h, and sql/log_event.cc
....
  class Log_event{
+   typedef unsigned char Byte;
  }
- void get_strlen_and_ptr(const char **src, ....
+ void get_strlen_and_ptr(const Log_event::Byte ** src, ....
....
```
i) Polymorphic

Figure 3. Examples of the patch categories. Simplified for illustration.

seems insufficient, so we consider such changes to be beyond our reach.

Macro Changes to macros, as in (h), are another case we currently don't deal with; we can't identify where in the code the change actually occurs. However, here it seems like slight assistance from the preprocessor should be sufficient to label where the patch has an effect, and therefore allow Δ execution.

Polymorphic As with macros, changing the type of something polymorphically (e.g. as in (i)) introduces scattered changes in the code far separated from the textual change. Again as with macros, the needed assistance from the compiler seems minor.

Complex Omitted from our examples, some patches are simply too complex or intrusive for Δ execution. Such patches involve many changes across hundreds or thousands of lines in multiple modules. Although we cannot deal with them, we would hope that the difficulty in reasoning about such patches (Torvalds 2000) and the availability of an easy mechanism for validating smaller changes would encourage programmers to avoid such patches.

2.2 Distribution of the categories

We summarize the distribution of patch types among each category in Table 2. Conditional changes dominate the number of patches, and there are also many stack effect patches. These and the other Δ executable changes comprise 77% of the changes. A further 6% of patches seem like they only need minor compiler support, implying that Δ execution should be effective for up to 83% of patches. This implies that there is a large potential to exploit the similarity between different versions, and that Δ execution is quite promising.

3. Delta Execution Idea

The patch study implies that the large bulk of execution during patch validation is completely redundant. Hence there seems little reason to actually have two separate runs. We could instead have only one context of execution, which applies the state updates to both copies. Further, since most of the state is also the same, we can maintain only one physical set of state. The small portions of execution where we need two executions streams and two sets of state can be treated as a special case. This is the key of delta execution.

Figure 2 shows Δ execution at a high level. Initially, we run only one execution, and have only one set of state. Runtime monitoring will tell us when we eventually access patched code (or *delta code*). This causes a split; execution continues with two contexts, each running a different version. I/O operations must be monitored, since they may differ and need validation. Also, while split the runtime tracks data writes, as the values may potentially differ, forming new *delta data*. After we merge, we must monitor for accesses to delta data, because it signifies we need to split again.

Eventually, the two separate executions should be merged together again. Any time the register states and instruction pointers of the two contexts are identical the two executions may be merged. This is because computers are deterministic state machines — once two contexts are identical they will change states in lockstep until either an instruction fetch or a data load causes an access to delta code or delta data. Hence, when each instance reaches the end of the patched code, we block them to compare to one another. If they match, merged execution will continue, with the runtime environment monitoring for further splits.

Patch statistics in categories											Δ-EXE %		
App.	refact	rare	stack	side	cond	sync	struct	macro	poly	comp	YES	W/ C	NO
Apache	5	8	16	10	31	0	7	5	0	3	76.7	83.3	16.7
MySQL	4	4	12	11	34	2	7	2	2	3	78.3	83.3	16.7
OpenSSL	6	7	14	7	28	3	1	3	0	4	81.7	88.3	11.7
Squid	2	6	11	9	32	0	7	5	1	5	73.3	78.3	21.7

Table 2. Patch characteristic study summary."W/ C" indicates YES with compiler support.

Of course, all of this isn't free. It may be that Δ execution is unsuitable for certain patches. Furthermore, the underlying delta execution mechanism must cost something. Finally, Δ execution is not trivial to actually implement. Specifically, we must address the following challenges:

- **Splitting.** Whenever the execution would diverge between the original and the patched version, we must split. We need to identify when such divergence would occur, begin two instruction streams, and arrange for the eventual reunification.
- **Running split.** When running split, we want to efficiently run two different instruction streams. Further, any data writes may be different between the two versions; hence we must track such writes, and ensure that any reads receive the correct result.
- **Merging.** Without merging, most of the benefit would be lost; however, we must somehow detect that the executions are again identical, identify which data is shared between the two instances, join the two streams, and arrange for detecting when we should split again.
- **Dealing with system calls and I/O.** Both versions will want to perform system operations; we must minimize redundant work, ensure that both versions receive the data they ought if they were alone, and at the same time prevent the non-production version from becoming externally visible.
- **Maximizing merged execution.** Δ execution's benefits come from the merged execution; we want to minimize the amount of time we spend split, eliminate unnecessary splits, and attempt to prevent minor differences in the two executions from blowing up into irreconcilable splits.
- **Delta execution and threads.** Threads pose their own special challenges, as detailed in Section 4.3.

4. Implementation and Issues

The previous section describes the overall idea of Δ execution; in this section we discuss the practical issues surrounding implementation.

4.1 Basic Delta execution

As described previously, there are several basic issues to address. Namely, splitting & deciding when to split, running split, and then how to merge again:

Splitting There are two reasons to split: delta code and delta data. For delta code, we generate a list of patched functions, and then use dynamic instrumentation to insert a split at the beginning of them. For delta data, we use `mprotect` to deny access to pages containing delta data. Using the page protection hardware allows us to avoid the expense of instrumenting every memory access.

To actually split, we use the `fork` system call. `Fork` creates two copies of the current program and arranges for copy-on-write sharing of pages: exactly what we need. If the split was caused by delta code, we set the instruction pointer of the child instance[1] to the patched version of the function. Regardless of why we split, we also copy any saved delta data into place for the child instance (the original version's delta data being stored in the proper place already). We now have two instances, one original running the original code with the original data, and one modified running (if applicable) modified code with modified data.

Running split Once set up, running split is fairly straightforward. `Fork` takes care of much of the work, however we still need to track delta state ourselves. When splitting, we use `mprotect` to re-enable access to delta pages (pages containing Δ data), and then to deny write access to all non-delta pages (pages which are identical in the two versions). Hence we will be notified of any writes which may create new delta data. We can record such writes and then reallow write access for those pages. `Fork` then takes care of actually copying the page for us.

Merging If we never merge after splitting, then we may as well have just run two copies. There are two issues with merging; when and how. We try merging at every function return, if the nesting level is less than what it was when we split. As the patch study showed, we can generally expect separate functions not to affect one another, so this works quite well as a heuristic. As future work, we may consider dynamically adapting where to merge, based on the success of previous merge attempts. As for how, when either instance hits a merge point (e.g. end of a function), we block them until the other instance is there as well. If their processor states are identical, we compare the pages we recorded as being written to and see if they really are different; if so we save them as Δ pages. Once the child's copy of delta pages are saved, we terminate the child, and then use `mprotect` to re-

[1] By arbitrary convention, the child is always the patched version.

allow access to all pages *except* the delta pages, which we begin monitoring. Finally, we continue with merged execution.

4.2 Advanced Delta Execution

Although the basic implementation of delta execution will work well for simple patches in simple programs, it isn't capable of handling more complex patches or programs. Here we address the further features needed for such cases.

Dealing with I/O During split execution, I/O writes from the child instance must be verified and sandboxed. Further, network reads are non-idempotent; data read by one process won't be there for the other. Hence, the Δ execution runtime must instrument all calls to `read` and `write` to manage the I/O operations. We can then 1) validate the modified copy's writes, 2) perform non-idempotent reads (e.g. from sockets or fifos) only once and forward the results to both instances, and 3) ensure that these relative expensive operations are done only once even for idempotent I/O calls. Also, because an I/O syscall could target delta data, even in merged mode we monitor the I/O operations and perform the necessary validation.

One issue with I/O that we don't address is that if the verification instance sends a remote procedure request that the live instance does not; we can't share the results because the visible instance didn't do it. This can be addressed by maintaining a verification instance of any possible RPC targets, and is a problem that faces any on-line validation system.

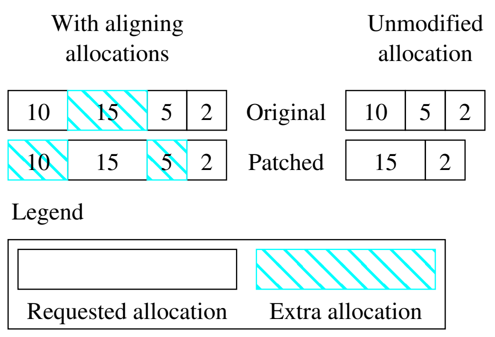
Maximizing merged execution To maximize the benefit of Δ execution, we want to maximize the time spent merged. Hence, we must minimize the amount of time we spend split. One source of unnecessary splits is false delta data. That is, data which is literally different and will be detected as delta data, but is actually semantically identical. For example, the area of the stack above the current frame, and freed heap objects, are dead state; they won't have any further effect on our execution. If we "scrub" this state after it is no longer needed, we can make additional pages exactly identical, and reduce the amount of delta data.

Another source of false delta data is from differing memory allocations. If the original and the modified executions allocate different amounts of memory, the heaps will become different, and subsequent allocations will be shifted between the two. This will cause a large divergence in the amount of detected delta data. Worse, this can cause `malloc`'s internal data structures to differ between the two, and any allocation or deallocation in the future will cause a split. The patch in Squid (see Section 6) suffers from this problem particularly badly; without help, the entire heap quickly becomes delta data.

In order to reduce the amount of false delta data from differing heap allocations, we instrument all calls to `malloc` and `free` that occur during split execution. As shown in Figure 4.a, if either instance makes an allocation different from the other, we will make false allocations for both. For example, suppose the original allocates 10, 5, and then 2 bytes, while the patched version allocates 15 and then 2 bytes. By inserting a false 15 byte allocation in the original, and false 10 and 5 byte allocations in the patched instance, we can force the two versions' heaps to line up. Also, by calling all memory management routines in the same order for both instances, we prevent `malloc`'s internal data structures from becoming delta data. By intercepting `free` calls, we track who has `free`'ed what, and only actually free when there are no more valid allocations. For Squid, these modifications allow efficient Δ execution.

While this reduces the impact of heap changes on Δ execution, it can cause some problems. Suppose the patch was trying to fix a memory leak. We will end up replicating the leak in both instances. Further, by changing the layout of the heap, we may cause some memory bugs to occur differently. Since we are targeting type-unsafe C programs, this is the best trade-off we can manage. If we were in a type-safe environment, such as Java, we could ignore the binary value of references and instead track if the underlying objects were "the same". This remains a possibility for future work.

Reducing split/merge overhead If a patch lies on a commonly taken execution path, we may end up needing to split and merge quite often. Although `fork()` as a splitting mechanism is relatively cheap, the time spent synchronizing during a merge is prohibitively expensive. The patch in ATPhttpd shown in Figure 3.c, for instance, is run multiple times per request, require multiple expensive merge operations.



(a) Increasing heap similarity

```
DELTA_START
//This is common code
D_ORIG_START
//This was removed in the patch
D_ORIG_END
D_MOD_START
//This was added in the patch
D_MOD_END
DELTA_STOP
```

(b) Supporting small delta segments

Figure 4. Techniques to maximize merged execution.

To alleviate this, we propose an instrumentation based split/merge. First, save the processor context, and run the unmodified version. Rather than use `fork()` to copy pages which are written to, the signal handler can copy the pages, as needed, when taking faults from `mprotect`. When a merge point is hit, save any modified pages, save the processor state, and then roll back to where we split. Then run the modified version, restoring delta state from the copy kept in the runtime environment's memory block. When this second run reaches the same merge point, we do the necessary comparisons, save any true delta data, and continue on with merged execution as usual. Although this won't run the two instances simultaneously, this should be more than made up for by avoiding synchronization time between the child and the parent.

Handling small deltas in large functions Sometimes, a small patch involves only one branch path of a large function. If we flag the entire function as delta code, we will cause unnecessary splits. To avoid this, we implemented a mechanism to support smaller segments of delta code. We label the areas of delta code, and what was was added/subtracted, by inserting macros in the source, as illustrated in Figure 4.b. DELTA_START and DELTA_STOP markers wrap around the beginning and end of segments containing delta code. The runtime environment instruments the DELTA_START markers with splits, and the DELTA_STOP markers with merges. D_ORIG_START and D_ORIG_END markers wrap around code which was removed in the patch ("-" lines in a diff). The runtime will skip these segments for the instance running the modified code. Likewise, D_MOD_START and D_MOD_END wrap around added code ("+" lines in a diff). Given the diff of a patch, it is trivial to decide where to place the markers, and could easily be automated by the compiler.

Competitive analysis for worst case situation In some cases, Δ execution may work poorly. It may also be that most of the execution between the two versions actually is different. This may be due to a trickle-down effect in the data (where one change begets another and then another), or perhaps due to the differing code leading to large differences in execution. Alternatively, the executions could be quite similar, but because the change is in a hotspot most of the time is lost to splitting and merging overhead. To alleviate such cases, we have a competitive fallback mechanism. We divide the execution into epochs, and monitor how much of each epoch is spent split, merged, splitting, and merging. Given an estimate of how inefficient running two instances side-by-side is, we can calculate if during the epoch we would have been better off without Δ execution. More formally, we should continue with Δ execution only if

$$\frac{\epsilon \cdot time\_merged + time\_split}{epoch\_length} \geq 1$$

where ϵ is the slowdown imposed by running two instances. If this inequality does not hold for 3 epochs in a row, we dynamically remove the Δ execution runtime environment, and revert to a fallback side-by-side validation. Later, if the load on the system is reduced, Δ execution can be re-enabled for high-quality validation.

Type changes A final issue which we are incapable of completely addressing is type changes. While changes in the layout of the stack are acceptable, type changes, especially of heap objects, are not well handled by our implementation. The simple change of adding a field to a structure completely breaks our implementation. However, although it is difficult, changes in type can be dealt with given sufficient language or compiler support (Segal and Frieder 1993). For arbitrary C programs, however, it is much more difficult, and we simply do not support such patches.

4.3 Threads

Given the current trend of increasing numbers of multicore processors, we would be remiss if we did not address the issues raised by threads. Beyond the increased complexity of writing thread safe instrumentation, threads raise several difficulties. First, threads are more difficult to split because the `fork()` system call doesn't duplicate threads. Further, we must deal with thread creation or destruction during split execution.

There are 3 ways to deal with `fork()` not duplicating threads. One can create a modified `fork()` call, which does duplicate threads. This has the advantage of supporting the widest variety of changes and programs. The disadvantage is requiring a modified system call; this is fairly intrusive. Alternatively, one can use instrumentation to recreate the threads. By directing a signal at each specific thread, on can trap them in a barrier. Then, the splitting thread can copy their execution contexts and *then* fork. The parent can then resume the stopped threads, while in the child process, we recreate all of the threads before continuing. Although this does not require modifications to the kernel, it is much more expensive. Finally one can temporarily disable other threads for the duration of split execution. Again, a signal can be used to pause the other threads; we don't have to recreate them. This is simple, inexpensive, and minimally intrusive. The downside is the potential for deadlock if the thread which caused the split needs a resource which has been locked by one of the suspended threads. Such deadlock situations can be avoided by temporarily merging execution, executing the thread that holds the resource, and then splitting again. For our primary implementation, we have chosen this option. We have never observed such deadlock in practice, and we can avoid requiring changes to the kernel.

Thread creation and destruction during delta execution is more complex. If both versions create (or destroy) a thread, then issues are minimized. However, if only one version does, then we will have a mismatch in the number of threads. The number of live threads is in some ways state intrinsic to the entire process, and may prevent merging. However, aside from contrived cases (e.g. changing a `MAX_THREADS`

Benchmark	Program	Change Description	Baseline	DE
crafty	Chess Program	Code refactoring	fails–kibitzes differ	pass
raytrace	Raytracer	Fixed bug in result reporting	large fail–nondeterminism	pass
tar	Archive Utility	Fixed incremental archiving	pass	pass
Apache 1	Web Server	Fixed overflow in mod_alias	fails–randomized etags	pass
Apache 2	Web Server	Fixed overflow in mime parser	fails–randomized etags	pass
ATPhttpd	Web Server	Fixed overflow in HTTP parsing	small fails–timestamps	pass
DNSCache	DNS Cache	Behavior change	fails–timing & random Tx IDs	pass
MySQL 5.0	Database Server	Extra permission checks	large fail–nondeterminism	pass
OpenSSL	Security Library	Added bug in TLS handling	MAJOR fail–cannot validate	pass
squid	Web Cache	Fixed overflow in FTP parsing	pass	pass

Table 3. Test applications and the effectiveness of their validation for running twice and delta execution. The baseline validation fails 8 of the 10 applications when they should pass; 3 of these (MySQL, OpenSSL, & raytrace) fail badly enough to be considered unvalidatable.

variable) such a situation is extremely rare; if it were to happen we can fail-back to side-by-side validation.

5. Methodology

For our evaluation, we implemented both Δ execution and traditional side-by-side on-line validation. For Δ execution, we used Pin (Luk et al. 2005), a dynamic instrumentation tool, to insert split/merge, to add appropriate signal handlers, to intercept system calls, etc. Our traditional validation is implemented as a network proxy in front of two separate instances (for network applications) or as a script wrapping two calls and verification (for command line applications).

Table 3 shows the test cases we used, including the application and a short description of the change. We tested 10 different applications, 7 of which were server programs. Further, 5 of the applications can use multiple cores. The workloads were as close to standard benchmark workloads as possible (e.g. the included workload from SPEC for crafty). However since the benchmark versions aren't buggy, and since the benchmark workloads won't exercise patches, these are not the standard benchmarks. All of the experiments were run on a 2-way SMP system with 2.4 GHz Pentium 4 Xeons and 2.5 GB of memory. The network was gigabit, and (as appropriate) the client machine was identically configured to the server machine.

For our performance evaluation, we compared the performance of three cases: (1) performing validation by running two copies side by side; (2) performing validation by running two copies side by side and using a null pintool, which performs no true work but exposes the underlying overhead of Pin's dynamic recompilation and code cache mechanism; and (3) performing validation using delta execution. All of our results are normalized to the performance of the same workload for one non-validated instance running in isolation.

6. Experimental Results

6.1 Effectiveness in Patch Validation

Table 3 shows the false positives generated during on-line validation. The baseline validation has trouble with 8 of the 10 runs. Only tar and Squid pass completely cleanly. Five applications file validation in smaller ways. Crafty continually prints what it is thinking; this differs slightly from run to run due to timing effects. For both Apaches, the randomly generated "Etag" field in the HTTP response (used for caching) fools the baseline into failing the patch. ATPhttpd occasionally fails because of time stamp differences, but for the most part the outputs are the same. Finally, DNSCache is sensitive to timing, and uses randomized transaction IDs; both of these cause differences in most of its replies. For these five failures, it wouldn't be insurmountable to create a detector to determine if the difference is a false positive or not. However, for MySQL, OpenSSL, and raytrace, this is not the case. Due to non-determinism in the thread interleaving, the output of both raytrace and MySQL will differ from run to run, even though it remains correct. Verifying that the output is still correct is difficult.

OpenSSL is hopeless for the baseline validation. The SSL handshake exchanges random numbers between the client and the server; since the client only sees the random numbers from the production instance, the replies it sends seem nonsensical to the testing instance. The test instance, therefore, reports an error and stops running; indeed, the proxy appears to be a man-in-the-middle attack, which SSL is supposed to detect. *Due to the non-determinism involved, the baseline validation does not work at all.*

In contrast with the baseline validation, delta execution correctly identifies all of the passing runs. Since the bulk of the execution occurs only once for "both" copies, delta execution ensures that non-deterministic effects don't cause differences in the output. Hence validation is greatly simplified.

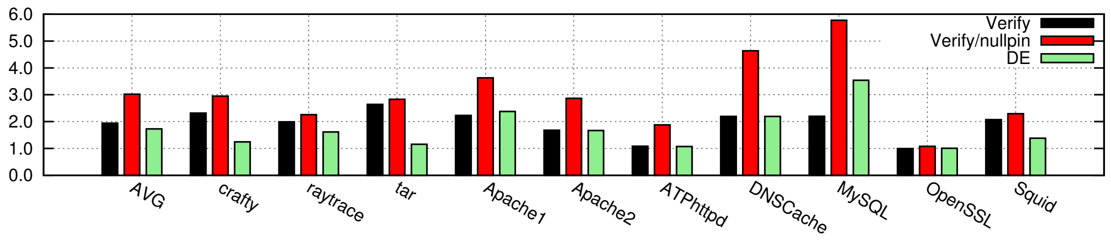


Figure 5. Overhead of validation: Δ execution (DE) compared to on-line validation with side-by-side and side-by-side running a null pintool. 1x is normalized to the performance of a single, unvalidated instance of the application.

6.2 Performance

Figure 5 shows the results of running delta execution. For crafty, raytrace, tar, and Squid, delta execution shows lower overhead than running twice, even after paying the cost of instrumentation. The best result is tar, for which Δ execution is 128% faster than the side-by-side validation. Overall, Δ execution is 12% faster. If the overhead of Pin is taken into account, Δ execution does much better; every application is faster using Δ execution, for an overall average increase in performance of 74%.

The "black sheep" performance-wise is MySQL. Δ execution is 37% slower than the side-by-side validation. Further, our adaptive mechanism fails to notice this. However, Δ execution is still 63% faster than side-by-side with nullpin. It turns out that Pin imposes a very large overhead even without instrumentation for MySQL, running 3.23 times more slowly even with nullpin. Δ execution is only 9.6% slower than this. Since the slowdown comes from the instrumentation, our adaptive mechanism believes Δ execution is doing well, and doesn't fail over.

Another interesting thing is the high efficiency of validation of ATPhttpd and OpenSSL. Both applications are CPU bound, yet neither program is capable of taking advantage of the multiple cores they had available. Hence, when running two copies side-by-side, there is an unusually low level of overhead. Indeed, because OpenSSL detects the network proxy as a man in the middle attack, *the second copy doesn't even perform any work*, and aborts the connection prematurely.

6.3 Detailed performance characteristics

Table 4 shows a detailed listing of where each program spent its time while running under delta execution. 8 of the 10 applications spend most of their time merged. The exceptions are tar (45% merged) and ATPhttpd (only 12% merged). Tar is especially interesting because it in fact performs better under Δ execution than any other application. Yet that tar spends 10.5% of its runtime split, and 40.6% of its runtime merging would imply that tar should have terrible performance. It turns out that most of the contention in tar is due not to the CPU but due to I/O. Although a large amount of CPU cycles are spend in merging, most of that time would have otherwise been spent waiting for the disk. Further, although tar has the most time spent split, most of that time is composed of waiting for I/O. Even when split, delta execution will issue only one I/O operation to the system if both versions are issuing the same operation. This allows tar to perform well under delta execution despite its large amount of split execution and merge overhead.

ATPhttpd, on the other hand, has the highest number of splits per second, at 19.1. On average among all of the benchmarks, a split takes 9.81 ms and a merge takes 105.9 ms. At 19 split/merges per second, ATPhttpd should spend 2.2 seconds per second either splitting or merging. Clearly it would make no forward progress at all if it weren't for the fact that it takes far less time per merge than average, only 35.3 ms. Since ATPhttpd is completely serial it does not suffer from scheduling contention while merging. Even so, ATPhttpd still spends over two thirds of its running time in merging back together. Unfortunately, although the patch in ATPhttpd does not change the path of execution in nearly all cases, the patch is called a minimum of *three times per request*. Cases like ATPhttpd motivate working on more efficient methods of merging.

7. Related Work

7.1 Systems administration

As "bits don't rot", we expect software systems to break only when something changes. We are far from the first to address this. Lowell et. al.'s devirtualizable virtual machines (Lowell et al. 2004) are thin VMs which may be inserted and removed as necessary; maintenance can be performed on a dynamic VM instance which can be then be rolled into production with minimal downtime. Although the primary goal is to limit downtime, the maintenance environment is turned into the production environment, allowing testing in an environment which is identical to what the new production environment will be. Unfortunately, validation on a full workload will still involve competition for resources with the production environment, and the previously discussed difficulties in actually validating correctness still stand. Similarly, Nagaraja et. al. measured real operator errors during various administrative tasks (Nagaraja et al. 2004). Based on their findings, they built a validation system which supported both on-line ("replica-based") and off-line ("trace-based") vali-

	crafty	raytrace	tar	Apache1	Apache2	ATPhttpd	DNSCache	MySQL	OpenSSL	squid
splits/sec	.005	.037	5.36	.368	6.560	19.10	9.638	.520	11.700	.903
%merged	99.996	99.134	45.40	94.500	72.900	12.10	55.691	87.827	59.500	88.200
%split	.001	.696	10.50	.002	.081	3.85	2.164	5.122	.229	.358
%splitting	.011	.130	3.08	.072	1.800	16.30	17.178	.469	6.290	0.896
%merging	.035	.055	40.60	2.420	25.20	67.60	24.917	7.440	33.900	10.500

Table 4. Detailed accounting of where time was spent in each application. Columns may not sum to 100% due to rounding.

dation. A "shunt" can capture requests and replies; the requests can be sent to a test instance which is tested against the shunted replies. Their capability to test against the full live workload makes the result of the validation trustworthy; however, they measure nearly a doubling of CPU usage while performing validation. Especially due to the trend of consolidating multiple servers onto one physical system with virtual machines, this CPU overhead is troublesome[2]. Additionally, they discuss the problems of non-determinism causing spurious differences in response.

7.2 Model checking

As we mention in previous work (Zhou et al. 2007), another use for delta execution is model checking (d'Amorim et al. 2007). By exhaustively exploring program states, model checking will expose bugs prior to release. Unlike in this work, (d'Amorim et al. 2007) focuses on running one instance of a program on many different inputs simultaneously. Modifications to the Java Path Finder model checker allow thousands of executions to run while sharing large amounts of state and execution. If only a few different executions are run simultaneously, (d'Amorim et al. 2007) will not share sufficient execution between different instances to overcome their base overhead. That it works terribly for validation, which has only two instances, is unsurprising; system extensions to support running two versions of x86 binaries is fundamentally different from JVM support for model checking.

7.3 Differences in software versions

The efficiency of delta execution relies on the observation that between versions, software remains mostly the same. If different versions of software were highly different, then all of the execution would be split execution. Fortunately, this is not the case. Work in binary differencing (Baker et al. 1999) and binary matching (Wang and Pierce 2000) shows that executables can be highly similar between versions. Further, although we use the compiler to create a binary with both the original and changed code linked in, binary differencing could extract delta code without the compiler's assistance. This would allow delta execution even if the vendor is unwilling to supply binaries with dual-versions of changed code. This remains as our future work.

In dynamic software update (e.g. (Hicks et al. 2001; Makris and Ryu 2007)), a program may be patched while it is still running. Of particular interest to us are procedure-based dynamic update systems, where individual procedures within a program may be updated. The PODUS system is an example of such (Segal and Frieder 1993). Although "any program can be so poorly written that it cannot be dynamically updated", most well-structured systems are suitable. During the update, two different versions will be resident at once. This is similar to how two versions are available in delta execution. However, dynamic update provides no checking that the two versions behave similarly, nor is it the intent that the two versions co-exist for any significant length of time. Also, unlike in our implementation of delta execution, the new version does not have to be available when the old version is compiled. Using the techniques of dynamic update, delta execution could be extended to support beginning validation without stopping the old version and starting a merged version of a program. Of course, the issues and limitations that dynamic-update systems suffer would then apply. Some (e.g. difficulty in changing data structures) currently apply to delta execution, while others (e.g. changes cannot be made at a granularity smaller than a procedure) would be more limiting than what delta execution currently supports.

Another technique similar to dynamic software update is band-aid patching (Sidiroglou et al. 2007). Band-aid patching will run the old and new versions of patched code in sequence. It then must immediately determine which version to use. The unused instance is then squashed; unfortunately this prevents dealing with faults with even short latent periods. Hence band-aid patching only handles cases where the change is isolated to one function, unlike Δ execution, which provides two logical versions throughout arbitrary lengths of time, and detects problems that require interaction between disparate code segments.

8. Conclusions

Given the likelihood of a change to a system causing a failure, the suspicion with which users and administrators view change is well warranted. Our proposed mechanism, Δ execution, allows them to assuage their concerns with on-line validation with their own workloads.

Our basic implementation of Δ execution reduces the overhead of on-line validation. It also reduces the spurious differences which frustrate traditional on-line validation, be they from thread interleavings, timing events, random numbers, or another source. Further, our study of real patches

[2] Client-perceived overhead was not measured

indicates that Δ execution should be applicable to a wide variety of cases.

Of course, although we have demonstrated a first practical implementation of Δ execution for validating program changes, there is more to be done. Changes that result in new types are not well dealt with, and require further language support, while some complex threading behaviors are not fully resolved. We would like to further explore methods for keeping the overhead better in check, and we hope to widen the applicability of Δ execution to include not only patches but configuration changes as well.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This paper is based upon work supported in part by the National Science Foundation under Grant Nos. 0347854, 0615372, and 0720743, and by Intel Gift grants.

References

Brenda S. Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999.

Rob Barrett, Paul P. Maglio, Eser Kandogan, and John Bailey. Usable autonomic computing systems: The administrator's perspective. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 18–26. IEEE Computer Society, 2004.

Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime, 2002. In *Proceedings of the 16th USENIX Systems Administration Conference (LISA'02)*, 2002.

Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen. A trend analysis of exploitations. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 214, Washington, DC, USA, 2001. IEEE Computer Society.

CERT. Cert statistics. http://www.cert.org/ stats/ cert_stats.html.

Jonathan E. Cook and Jeffrey A. Dage. Highly reliable upgrading of components. In *International Conference on Software Engineering*, pages 203–212, 1999.

Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The cracker patch choice: An analysis of post hoc security techniques. In *Proceedings of the National Information Systems Security Conference (NISSC)*, Oct 2000.

Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta execution for efficient state-space exploration of object-oriented programds. In *ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 13–23, New York, NY, USA, 2001. ACM.

Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *SIGOPS Oper. Syst. Rev.*, 39 (5):91–104, 2005.

David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ASPLOS '04*, 39(11):211–223, 2004.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace a nd Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 327–340, New York, NY, USA, 2007. ACM.

Evan Marcus and Hal Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

Paul McDougall. Microsoft pulls buggy Windows Vista SP1 files. InformationWeek, Feb 2008. http://www.informationweek.com/ story/ showArticle.jhtml?articleID=206800819.

Microsoft. Revamping the microsoft security bulletin release process, Oct 2003. http://www.microsoft.com/ technet/ security/ bulletin/ revsbwp.mspx.

Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, 2004.

National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, 2002.

Rob Pegoraro. Apple updates Leopard–again. The Washington Post, Feb 2008. http://blog.washingtonpost.com/ fasterforward/ 2008/ 02/ apple_updates_leopardagain.html.

Eric Rescorla. Security holes... who cares? In *Proceedings of the 12th USENIX Security Conference*, Aug 2003.

Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10(2):53–65, 1993.

Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis. Band-aid patching. In *HotDep'07: Proceedings of the 3rd Workshop on Hot Topics in System Dependability*. USENIX Association, 2007.

Linus Torvalds. Re: [rant] linux-irda status. Linux Kernel Mailing List, November 2000.

Zheng Wang and Ken Pierce. Bmat – a binary matching tool for stale profile propagation. *Instruction-Level Parallelism*, 2000.

Yuanyuan Zhou, Darko Marinov, William Sanders, Craig Zilles, Marcelo d'Amorim, Steven Lauterburg, Ryan M. Lefever, and Joseph Tucek. Delta execution for software reliability. In *HotDep'07: Proceedings of the 3rd Workshop on Hot Topics in System Dependability*. USENIX Association, 2007.