Feasibility of Mutable Replay for Automated Regression Testing of Security Updates

Ilia Kravets   Dan Tsafrir

Department of Computer Science
Technion – Israel Institute of Technology
{ilia,dan}@cs.technion.ac.il

Abstract

Applying software patches runs the risk of somehow breaking the system, so organizations often prefer to avoid updates or to defer them until extensive testing is performed. But unpatched software might be compromised, and extensive testing might be tedious and time and resource consuming. We consider alleviating the problem by utilizing a new type of "mutable" deterministic execution replay, which would allow us to (1) run the software before and after the patch and to (2) compare the behavior of the two versions in a manner that tolerates legitimate differences that arise from security patches. The potential advantages of this approach are that: it is automatic; it requires no programming skills; it is language-agnostic; it works without source code; and it can be applied while the system is running and serving real production workloads. In this work, we do not implement mutable replay; rather, we assess its *feasibility*. We do so by systematically studying about 200 security patches applied to the most popular Debian/Linux packages and by considering how mutable replay should be implemented so as to tolerate execution differences that such patches generate.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability

General Terms reliability, security

Keywords testing, record-replay, mutable replay

1. Introduction

After a software product is shipped, it typically goes into a maintenance phase whereby related software updates are made available form time to time [18]. Such updates should in principle have a positive effect, such as adding new features, improving performance, fixing bugs, etc. But in reality they often place a heavy burden on enterprise organizations, which need to worry about the possibility of updates somehow adversely affecting their systems. Indeed, very many patches have been reported to be buggy in their first release [13, 22, 23, 24]. Faulty software updates might translate, for example, into money losses or noncompliance with regulations [6], and so organizations often favor stability over the possible improvements brought by updates [6], preferring not to update if they can help it, or defer the updates until they are thoroughly tested.

There are, however, software updates that are hard to decline, notably fixes of critical bugs and security vulnerabilities. Leaving deployed software outdated with respect to such updates might lead to highly undesirable consequences [9, 14]. Nevertheless, many systems are not patched against security vulnerabilities for long periods of time [5, 7, 22]. And this state of affairs is worsened by the fact that malware authors regularly use security updates to reverse engineer system weaknesses and infer the corresponding exploits [2]. (It has even been shown that such inference can be automated [8].)

A standard procedure to lower the risks of upgrades is to have a "staging environment" to which the update is applied first. The system administrators are then responsible for ensuring that the staging environment operates correctly, in which case the update is also applied to the production environment. Microsoft Windows Server Update Services [19] supports this workflow [20], which is a recommended practice for update deployment [21]. The drawback of this approach is that the act of testing is left in the hands of the administrators, whereas testing is usually tedious, time consuming, often requires manual work, is prone to human error, and is limited by the available resources (manpower, equipment, etc.). Other, more advanced techniques for update testing have been proposed [12, 23, 24], but they usually require special support from the software vendor, such as source code or build environment access; we are not aware of such systems being used in production environments.

Another approach for dealing with faulty updates is to take the risk and later revert the update in case it is found to cause problems [4, 21]. This approach has obvious drawbacks. First, it might be initially hard to identify erroneous behavior, as the manifestation of the problem might be subtle and illusive. A second drawback is that it is not always possible to undo erroneous actions, and in some cases the damage can be significant even if the system has been successfully reverted. Finally, the revert procedure might induce unacceptable downtime.

In this paper we explore the feasibility of *mutable replay*, a complementary approach for testing security updates in an automatic manner. The hypothesis underlying mutable replay is that security patches tend to be small and unobtrusive. Therefore, in most cases, we contend that it might be possible to leverage existing deterministic record-replay mechanisms [17] to automatically test whether or not the security patches have introduced undesirable changes of behavior. We address mutable replay both generally (Section 2) and concretely as a possible extension of an existing deterministic replay system (Section 3). We note that the actual implementation of mutable replay is work-in-progress and is not described in this paper. Instead, in this work, we strive to keep the presentation generic and independent of specific implementation details. Our major contribution is a survey of more than 200 security updates that were issued for the most popular Linux/Debian packages over a period of more than two years. We utilize the results of this survey to identify what would be required from a deterministic execution record-replay mechanism if it is transformed to tolerate differences arising from security patches (Section 4). Finally, we compare the proposed approach to the related work (Section 5).

2. Utilizing Mutable Replay

2.1 Goal

When the behavior of two program versions—before and after a security patch is applied—diverges, then either (1) the old, unpatched version is right (exhibits correct behavior) and the new, patched version is wrong (exhibits incorrect behavior); or (2) the old version is wrong and the new version is right; or (3) both are right.[1] Counter-intuitively, the focus of this paper is *not* the first and second cases. Rather it is the third. This statement might be perceived as negating the paper's main motivation of helping clients to protect against faulty patches. But it does not.

The first aforementioned case means that a patch is faulty. And the second case means that the system is under attack (which should in principle be prevented by the patched version). We acknowledge outright that we have no general automatic way to tell these two cases apart, nor is it our goal to do so. Instead, we are satisfied by the mere prospect of being able to automatically report to clients that one of the cases 1–2 occurred, namely, that either their system is under attack or the patch they have applied to it is faulty. Such a report is expected to typically be highly valuable to clients, as knowing that this type of events occurred is much preferable than not knowing. Following such a report, clients would be able to further investigate to determine the exact cause of the event and then take the appropriate actions (initiate an intrusion mitigation procedure or conclude that a patch is harmful and reject it).

The focus of the paper is the third case—when both patched and unpatched versions are right—because detecting divergence between two programs is easy, if not trivial; the real challenge is determining whether a divergence is semantically meaningful and being able to tolerate the divergence if it is not, without alerting the user. Below we enumerate the most common changes caused by security patches that trigger divergences but do not change program semantics. We argue that a record-replay system that is able to quietly tolerate these changes is sufficient for implementing mutable replay that provides a valuable service to clients: reporting if/when an attack occurred or the patch is discovered to be faulty, or not reporting anything, which means the old and new versions have the same semantics thus far.

2.2 Usage Model, Performance, and Overhead

A mutable replay system has several possible deployment scenarios [10], including doing analysis at realtime, in an online manner. But in the context of this paper, to simplify, we focus on the scenario where the production (unpatched) system does not perform testing on its own or actively participate in coordinating such testing. Instead, it merely runs in recording mode, producing an activity log to be used by the mutable replay testing mechanism elsewhere; the log is then asynchronously transmitted to a different machine, where the testing takes place.

The overhead of recording a server's activity for the purpose of deterministic replay varies based on workload. For example, non-mutable recording can incur a 2.5% slowdown and generate an activity log growth rate as high as 1.9 MB/s [17]. Mutable replay might require more data to be logged and thus greater overhead. This overhead may perhaps be reduced using techniques such as multi-stage replay [11], which, in principle, can make the overhead of recording for the sake of mutable replay comparable to that of recording for non-mutable replay.

The performance of the testing machine might likewise vary depending on the actual workload; as a reference, the *speedup* reported for deterministic (non-mutable) replay in comparison to the record phase ranges from ~1x faster for CPU-intensive workloads to 70x faster for interactive workloads (the result of not having to wait for input) [17]. We note, however, that the performance of replaying is less of a concern in our usage model, because it is decoupled from the production environment. The cost of maintaining a testing environment can be amortized by reusing the same hardware to test several systems.

While mutable replay is likely most useful for organizations that have resources to deal with its report, we believe it can also be valuable in smaller setups. An organization with enough resources might utilize mutable replay as an initial screening process, directing human efforts to doing manual inspection for only those patches that failed the automatic testing. If an organization requires manual testing of certain specific scenarios, the latter can be recorded and tested against an updated version, automatically, through mutable replay. In smaller setups, e.g., when an individual user lacks resources and expertise to overcome the reported problems on her own, mutable replay can be used to get a quick indication of whether the upcoming updates require special precautions, such as performing a full backup, planning an additional downtime, or postponing the update [6] and requesting support from the software vendor.

3. Implementing Mutable Replay

As noted, the goal of mutable replay is to allow for automatic regression testing under realistic workloads. A canonical way to perform regression testing is to run two versions of the program—before and after the change—with the same input, and then examine the output. The testing procedure can be partitioned into three tasks: determine the input, feed it to both versions, and compare the output. In the context of mutable replay, the first and second tasks are accomplished by recording all the nondeterministic aspects of the execution of the unpatched program (e.g., input data, timing, order of accesses to shared resources) and reconstructing them during the execution of the patched version.

The problem is that the extensive work done on record-replay thus far [15, 16, 17, 25] aims for replay that accurately reflects the recording. Execution divergences of programs (as would inevitably occur due to the patches) are irrelevant in those systems; they are out of scope and are not allowed to occur by design. Conversely, to enable the mutable replay approach that we advocate, we need a system capable of tolerating some program behavior changes.

3.1 Extending SCRIBE, an Existing Replay Mechanism

Fortunately, we find that SCRIBE—an already existing deterministic replay mechanism integrated into the operating system kernel [17]—nearly meets our goals. We next provide a brief description of SCRIBE, directing the interested reader to the original paper for more details [17].

The SCRIBE mechanism allows its users to record and deterministically replay the execution of one or more processes running under Linux. SCRIBE works by intercepting system calls, non-deterministic hardware instructions (like `RDTSC`), and accesses to shared resources. During the record phase, SCRIBE stores the intercepted events in a log. Then, during replay, SCRIBE sequentially reads the events from the log and replays them in order. During replay, SCRIBE keeps both application and kernel state compatible with the associated recording. This property allows SCRIBE to reduce the log size by storing just enough information to steer the program replayed execution such that all missing information is supplied by the application or the kernel as the replay progresses. In addition, maintaining up-to-date state allows SCRIBE to switch to live execution at almost any point in the replay sequence, as most parts of the kernel do not distinguish between replay or normal execution mode.

[1] The case where both are wrong is irrelevant here.

```
#include <stdio.h>
#include <string.h>

int main()
{
  char secret[] = "1337", buf[5];
  char *msg[] = { "access granted\n",
                  "access denied\n" };
  int err;

  puts("Enter secret: ");
- scanf("%s", buf);
+ scanf("%4s", buf);
  err = strcmp(buf, secret)!=0;
  puts(msg[err]);
  return err;
}
```

Figure 1. Version divergence due to the above simplistic security fix can be be tolerated by SCRIBE, out of the box.

We note that the maintenance of updated kernel state is crucial for mutable replay, as the execution of the patched version might require some information that was not recorded in advance. Additionally, in some cases, it might even be required to update that state in order to proceed correctly.

SCRIBE ensures deterministic replay of asynchronous events (such as signal delivery) by postponing them until the occurrence of a synchronous event (such as a system call). When a pending asynchronous event awaits the occurrence of synchronous event, SCRIBE sets a timer to expire after a short, predetermined amount of time. Upon timer expiration, if no synchronous event has yet occurred, SCRIBE artificially generates one by taking a snapshot of the execution context and logging an event holding the content of the registers and a checksum of the writable memory. This event is delivered during replay when the program reaches the same exact state of the execution context. Alas, we suspect that such as approach is inapplicable for mutable replay, as a match between the recorded and the actual contexts might never occur due to the difference between the program versions. We are currently unable to propose a general solution with low overhead to this problem.

Since SCRIBE operates at the operating system abstraction level, it can already tolerate—out of the box—differences arising from trivial patches. Figure 1 demonstrates such a patch to a fictitious authentication program. The patch changes are marked with a minus and plus signs. Before applying the patch, the application is vulnerable to buffer overflow and consequently to arbitrary code execution (or at least to unlawful authentication bypassing). Yet, from SCRIBE's perspective, as long as the input is legal (no attack is launched), the events sequence remains the same, which means the simple patch does not change the program behavior (as viewed by SCRIBE), even though the memory content and execution are slightly different. Conversely, upon illegal input, the event sequences will diverge and SCRIBE will issue an error message reporting this fact (which is the desirable behavior in such cases).

4. Surveying Security Updates

To assess the feasibility and applicability of mutable replay in the presence of real security patches, we conduct a survey of the security updates delivered to the users of the Debian GNU/Linux distribution over a course of 2.5 years.

4.1 Methodology

We manually examine the source code of each patch and attempt to identify (1) how the change it induces will affect an OS-level replay mechanism and (2) how the mechanism should be modified so as to tolerate the change. We do not conduct actual program runs.

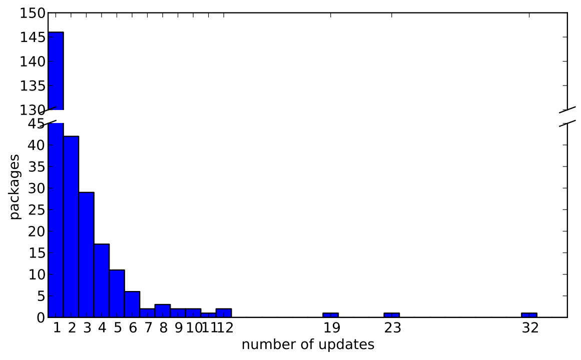

Figure 2. Histogram of packages based on the number of updates they received.

Terminology Debian organizes applications and libraries in packages. Our analysis is performed on source packages, which can be built and thereby generate one or more binary packages, which in turn can be downloaded and installed by end users. A source package consists of the upstream application/library source code (as released by the original software developers), Debian-specific patches, and build instructions and metadata such as the package description, its dependencies, and a textual changelog tracking its release history. It is not uncommon for Debian developers to include several patches in a single source package update, also denoted a "release". From this point onwards, we interchangeably use the terms "patch" to refer to an entire release or update; a package can have more than one release, and we do not unify those.

Debian Security Patches We analyze the history of security updates applied to Debian 5.0 (Lenny) from its release date in February 2009 until August 2011. Nearly all of this data is readily available on Debian's web and ftp sites, including most of the per-package releases (rather than just the most recent release). There are 280 Debian 5.0 packages that are categorized as having security updates. Of these, 81 (nearly 30%) have no other purpose but encapsulating shared libraries, and an additional 27 (nearly 10%) contain shared code to be used by other programs; thus an update to these 81+28=108 packages can potentially affect more than a single application or service. It is interesting to note that even in this small data set we find 14 updates that fix regression problems that were introduced by earlier releases, demonstrating the imperfection of the security update release process and supporting the reluctance of some system administrators to apply the patches.

Figure 2 shows the frequency of package updates over the 2.5 years that we have analyzed. Most packages received a single update, and only 21 packages received more than 5. Unsurprisingly, we observe that the most frequently updated packages are quite popular, including glibc and OpenSSL (libraries), Apache, Samba, BIND, and PostgreSQL (servers), and PHP, OpenOffice.org, and Firefox (desktop application). OpenSSL updates alone would, on average, require system administrators to test all services depending on OpenSSL encryption (web, email, etc.) every 12 weeks.

Patch Selection Out of the aforementioned 280 packages, we choose to manually inspect the 65 most popular. (Popularity is determined based on the number of installations of the packages as reported in the official Debian popularity contest website [1].) The number of security updates associated with the chosen 65 packages is 220. We manually analyze each and asses its potential to interference with mutable replay and cause a divergence.

Characterizing Divergence Likeliness Based on our inspection of the code, we approximate the likeliness of each patch to trigger a replay divergence. To this end, we associated a *likeliness level* with each patch; we use three levels: "usual", to denote that normal program usage would likely trigger a divergence; "unusual" to denote that normal program usage might trigger a divergence in some uncommon cases; and "rare" to denote that normal program usage is highly unlikely to trigger a divergence, unless specifically crafting a triggering condition, notably, attacking the system. When our level of understanding of the patch is insufficient to determine the likeliness of divergence, we conservatively assign a "usual" rating. An "unusual" divergence may involve running uncommon combination of the program components, configuration, and input. Arguably, "rare" divergences are of less interest because, if the system is under attack, then we actually want a divergence to occur and for the mutable replay mechanism to report it; otherwise, we contend that users are unlikely to encounter these divergence and that it is more important to focus on the common cases.

Characterizing Divergence Severity Orthogonally to the probability or likeliness of a patch to trigger a divergence, we would like to assess whether it is possible to tolerate this divergence using a mutable replay mechanism. Toleration might not be possible if the patch is too big or too intrusive or introduces inherent changes that make convergence impossible. Accordingly, we classify all the patches to those that are likely to be tolerated by a sophisticated-enough replay mechanism and those that probably deem replaying impossible regardless of the degree of sophistication.

Characterizing Techniques to Tolerate Divergence Finally, focusing on the patches that are classified as triggering a tolerable divergence, we would like to identify what it would take in order to make a replay mechanism sophisticated enough (mutable enough) so as to tolerate the corresponding divergence.

4.2 Results

Table 1 summarizes our findings regarding the severity of divergences, quantifying how many patches generate tolerable and intolerable divergences. We find that a mutable replaying mechanism may, in principle, tolerate divergences triggered by 75% of the patches, but would likely fail to do so for 4% of the patches that drastically change the behavior of the program. Another 15% of the updates are difficult to analyze due to their size, which is too big; we can probably safely assume that they introduce intolerable divergences as well. We observe, however, that while they are labeled and packaged by Debian maintainers as security patches, in fact, these large updates are typically new versions of software, and so the changes they introduce go far beyond what we would normally characterize as a security fix. (It seems reasonable to expect that more enterprise-friendly distributions, like Redhat, would not push such updates to clients as security patches.) The remaining 6% of the updates are small, but we are still unable to determine their impact on overall program behavior from just reading the code; for example, changing a mathematical computation deep within the call hierarchy may result in both tolerable and intolerable divergences.

We next go on to characterizing and grouping typical types of divergences caused by security patches. To exemplify, we utilize Figure 3 that shows an excerpt from a security fix of acpid — a daemon handling ACPI events in Linux. The fix introduces a new function, `acpid_close_dead_clients()`, which is used to close stale client connections that will no longer be used for communications. The fix also includes periodic invocation of this function from the server event loop (not shown), helping to reduce resource usage (notably, of file descriptors) and thereby fixing a denial of service vulnerability [3]. In the absence of an attack, from the user perspective, the daemon continues to function as usual after the upgrade. But the resulting changes are sufficient to cause several types of divergence: new system calls are added, both for inspecting (`poll`, `getsockopt`) and for modifying (`sigprocmask`, `close`) the kernel state; log output may result in additional `write` system calls that may occur unexpectedly due to application output buffering; and freeing memory may likewise change the heap layout and hence subsequent allocations performed by libc through `brk` or `sbrk`. We group these and other observed possible divergences based on their characteristics and, in some cases, we describe how to tolerate them during replay:

Memory Layout Change An updated program has a somewhat different memory layout, prompting changes in the program counter and in system call arguments that hold addresses, either temporarily or throughout the entire execution. The change may

tolerance	*updates*	*% of total*
tolerable	165	75%
intolerable	9	4%
unclear (large)	32	15%
unclear (logic)	14	6%
total	220	100%

Table 1. Severity of divergences cause by security patches.

```
// this function is called from the server event loop
void acpid_close_dead_clients(void)
{
  struct rule *p;

  // sigprocmask syscall: block HUP,TERM,QUIT,INT signals
  lock_rules();

  /* scan our client list */
  p = client_list.head;
  while (p) {
    struct rule *next = p->next;
    // poll syscall: check for POLLERR and POLLHUP events
    if (client_is_dead(p->action.fd)) {
      struct ucred cred;
      // write syscall: add log output
      acpid_log(LOG_NOTICE,
            "client %s has disconnected\n", p->origin);
      // no syscalls, just remove linked list element
      delist_rule(&client_list, p);
      // getsockopt syscall
      ud_get_peercred(p->action.fd, &cred);
      if (cred.uid != 0) {
        non_root_clients--;
      }
      // close syscall: close unneeded file descriptor
      close(p->action.fd);
      // (s)brk syscall: free heap memory
      free_rule(p);
    }
    p = next;
  }

  // sigprocmask syscall: allow HUP,TERM,QUIT,INT signals
  unlock_rules();
}
```

Figure 3. Annotated excerpt from a patch to the acpid daemon. The newly added `acpid_close_dead_clients()` function is called from the daemon's main event loop, periodically.

affect different memory segments, e.g., adding the function shown in Figure 3 to acpid affects its code segment, and line 18 affects its statically allocated data (even before it runs). The heap (line 29) and the stack are affected dynamically during program execution. In some cases memory allocation may change sufficiently to affect the pattern of memory management requests, such as `mmap` and `brk`.

Memory layout changes are widespread: only one of the updates that we analyzed was identified as completely preserving the memory layout (the whole patch was limited to a single constant change). Therefore, for a mutable replay mechanism, it is critical to tolerate this type of change so as to avoid replay divergence.

Read-Only System Calls A patched version can add or remove system calls querying kernel state (security updates more often add than remove such calls). See, for example, Figure 3, lines 14 and 22. Maintaining an up-to-date kernel state is useful for tolerating such changes. The kernel may also need to perform additional I/O to query filesystem-related information. If the required information is unavailable, the replay system can still try to emulate proper system call execution. For example, if a program queries the time, the returned value should be consistent with past and future queries.

Equivalent System Calls When interfacing with the kernel there are often more ways than one to achieve the same goal. For example, we can receive a socket message using either `recv` or `recvmsg`; or we can change the mode of a file using `chmod` or `open+fchmod +close`; or we can access the same inode using multiple file paths. Security patches often replace one way of interacting with the kernel with a different, seemingly equivalent way, because attackers tend to exploit the uncommon case where the two ways lead to different results. But as long as the system is not under attack, mutable replay mechanisms should be able to tolerate such changes.

Output Format Change Security updates often introduce a small change in the program's output, while preserving all other behavior aspects; for example, when a web application is modified to prevent a cross-site scripting attack. The replay mechanism cannot automatically assess the "correctness" of such a change. But it can certainly verify that the the associated divergence manifests itself exclusively in one place and that convergence is achieved immediately after. Combining this approach with a textual report that visually presents the output difference seems like a reasonable way to verify such patches. Albeit, meaningful differences can be presented only when the network connections are unencrypted.

Unimportant Output Change Certain types of output are oftentimes considered by users as unimportant, notably when directed at logs. See, for example, Figure 3, line 17. A mutable replay system should be expressive enough to allow its users to specify which I/O channels are unimportant and to silently ignore the associated differences that occur while testing. Such a specification can be as easy as generating a configuration file comprised of statements like "/dev/null" or "all files under /var/log".

Resource Name Change A patched application may utilize different resource names while keeping all other behavior aspects intact. For example, the name of a a temporary file path can be changed so as to reside in a safer location.

Temporary Change Security patches sometimes add system calls pairs that change kernel state temporarily, notably for blocking and unblocking signals, so as to prevent race conditions and other undesirable corner cases. See, for example, Figure 3, lines 7 and 35. While unlikely, if such a race occurs when recording the unpatched version, then replay would probably diverge.

Synchronization Synchronization changes, notably involving locks, may be considered as causing a temporary change in kernel state as in the previous paragraph. But mutual dependence of threads upon the execution order might make replay support more challenging so we separate the categories.

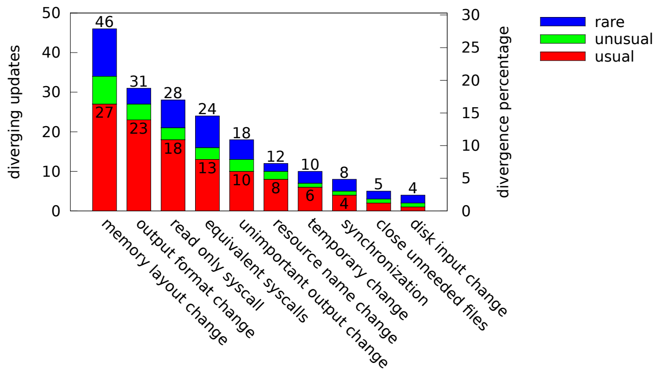

Figure 4. Summarizing the likeness level of the 165 tolerable patches. The X axis groups patches by the type of divergence they cause, and, for each bar, the Y axis shows the number of remaining patches that still cause divergence, assuming the type of divergence of that bar (and the bars to its left) are tolerated.

Closing Unneeded Files Security patches often improve upon the unpatched version by closing file descriptors that are no longer needed, thereby preventing resource exhaustion or data leakage. See, for example, Figure 3, line 27. This type of divergences should clearly be tolerated. The challenge for mutable replay arises due to later descriptor allocations, which would be numerically different than those recorded.

Disk Input Change Some security patches introduce changes not to the program, but rather to static data files stored on disk. For example, adding bogus certificates to a black list or comments to a configuration file. The associated patches we have examined should not create a divergence unless under attack.

Summary Figure 4 summarizes our findings regarding the 165 security patches labeled as tolerable (see Table 1). The X-axis denotes the divergence tolerance type/technique enumerated above, and the associated bars are ordered according to their contribution to the overall divergence decline, which is denoted by the Y-axis.

When a replay mechanism expects the memory layout of the patched and unpatched versions to be the same, without employing any tolerance techniques, then almost all updates may trigger a divergence. However, if the replay mechanism is modified to tolerate memory layout changes, the number of divergences drops to 46, which is comprised of 27 patches associated with a "usual" likeliness level, 7 patches associated with "unusual" likeliness level, and 12 patches associated with a "rare" likeliness level.

If the mutable replay mechanism is modified to tolerate both changes in the memory layout *and* in the output format, then the number of divergences drops to 31, comprised of 23 usual patches, 4 unusual, and 4 that have rare likeliness.

When further adding toleration to read-only system calls, the overall number of divergences is reduced to 28, yet, interestingly, the number of patches that have rare likeliness level (within these 28) gets bigger in comparison to the previous bar. This growth occurs because some updates that previously triggered divergences in a non-rare cases, now only do so in rare cases.

Our analysis indicates that a full implementation of the three first techniques (leftmost on the graph) is sufficient to tolerate 87% of all tolerable divergences in non-rare cases.

5. Comparing Mutable Replay to Delta Execution

The closest study related to our work is the "delta execution" project by Tucek et al. [24], which aims to solve the problem we too attempt to solve, namely, to alert users if security patches change the program behavior under normal (non-attack) conditions.

Briefly, when possible, delta execution simulates the execution of the two versions—patched and unpatched—using only one "merged" execution context. As long as the two versions execute the same code and access the same data, the delta execution mechanism runs only one merged version. When a different instruction is encountered, or when the same instruction accesses different data, the merged execution is split into two processes. Then, when the two split processes return to run the same code, they are merged again into one, while preserving data differences for a later split.

Delta execution has a two advantages over mutable replay. First, it is implemented in user space, and hence it is more portable. And second, in some workload/patch combinations, it may result in a single physical execution process for most of the testing period, thereby lowering the overhead. (On the other hand, the repeated action of splitting the unified context, running in split mode, and then merging again, might be expensive; and with delta execution testing can only be performed in an online manner, requiring both the production version and the test version to run on the same machine, which adds overhead to the critical path.)

A main drawback of delta execution is that it requires that most of the process's memory image will remain the same across the patched and unpatched versions. This means, for example, that: (1) updates to macros, inline functions, and structure (e.g., adding new struct member) are handled poorly or not at all; that (2) the approach is unfriendly to updates produced using a different compilation process, e.g., when using different compilers, optimizations, or debug/release mode; and that (3) the heap memory of both versions needs to be similar, which, according to our survey, might not be the case. Replacing the `malloc` and `free` functions might alleviate the latter problem, but only partially.

Importantly, delta execution lacks knowledge about system call semantics and hence will always declare divergence in the face of changed system calls. It may be possible to extend the delta execution framework with appropriate wrappers for all system calls, but doing so effectively translates to reimplementing much of the suggested mutable replay functionality.

Finally, delta execution has a serious limitation in that it requires source code access. The delta execution authors suggest that this limitation can be eliminated using binary differencing techniques.

References

[1] Debian popularity contest. `http://popcon.debian.org/`.

[2] Hackers piggyback on windows patches. *BBC*, Feb. 2004.

[3] CVE-2009-0798. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0798`, 2009.

[4] rPath. `http://www.rpath.com/`, 2011.

[5] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: a case study analysis. *Computer*, 33(12):52– 59, Dec. 2000.

[6] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the application of security patches for optimal uptime. In *Proceedings of 16th Systems Administration Conference*, volume 2 of *LISA '02*, pages 233–242, 2002.

[7] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen. A trend analysis of exploitations. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 214. IEEE Computer Society, 2001.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE Computer Society, 2008.

[9] K. Campbell, L. A. Gordon, M. P. Loeb, and L. Zhou. The economic cost of publicly announced information security breaches: empirical evidence from the stock market. *J. Comput. Secur.*, 11(3):431–448, Apr. 2003.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, Boston, Massachusetts, 2008. USENIX Association.

[11] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '10*, page 13, Pittsburgh, Pennsylvania, USA, 2010.

[12] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 203–212, New York, NY, USA, 1999. ACM.

[13] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The cracker patch choice: An analysis of post hoc security techniques. In *Proceedings of the 23rd National Information Systems Security Conference (NISSC)*, pages 16–19, 2000.

[14] D. A. Dittrich. Developing an effective incident cost analysis mechanism. `http://www.symantec.com/connect/articles/developing-effective-incident-cost-analysis-mechanism`, 2002.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 211–224, Boston, Massachusetts, 2002. ACM.

[16] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[17] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM.

[18] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[19] Microsoft. Windows server update services.

[20] Microsoft. Software update services deployment white paper, Jan. 2004.

[21] Microsoft. Update management process. phase 3 - evaluate and plan. `http://technet.microsoft.com/en-us/library/cc700840.aspx`, June 2007.

[22] E. Rescorla. Security holes... who cares? In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[23] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007. USENIX Association.

[24] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[25] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, San Diego, California, 2003. ACM.