Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting

Anthony Saieva, Gail Kaiser

PII: S0164-1212(22)00105-4
DOI: https://doi.org/10.1016/j.jss.2022.111381
Reference: JSS 111381

To appear in: The Journal of Systems & Software

Received date: 5 March 2021
Revised date: 16 March 2022
Accepted date: 20 May 2022


This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2022 Elsevier Inc. All rights reserved.
Update with Care: Testing Candidate Patches and Integrating Selective Updates

Anthony Saieva*, Gail Kaiser

Columbia University
Department of Computer Science
530 W 120th Street, New York, NY 10027

*ant@cs.columbia.edu
†kaiser@cs.columbia.edu (Corresponding author)
Update with Care: Testing Candidate Bug Fixes and Integrating Selective Updates through Binary Rewriting

Anthony Saieva\textsuperscript{a}, Gail Kaiser\textsuperscript{b}

\textsuperscript{a}Columbia University
\textsuperscript{b}Department of Computer Science
New York, NY 10027

\textsuperscript{a}ant@cs.columbia.edu
\textsuperscript{b}kaiser@cs.columbia.edu

Abstract

Enterprise software updates depend on the interaction between user and developer organizations. This interaction becomes especially complex when a single developer organization writes software that services hundreds of different user organizations. Miscommunication during patching and deployment efforts lead to insecure or malfunctioning software installations. While developers oversee the code, the update process starts and ends outside their control. Since developer test suites may fail to capture buggy behavior finding and fixing these bugs starts with user generated bug reports and 3rd party disclosures. The process ends when the fixed code is deployed in production. Any friction between user, and developer results in a delay patching critical bugs.

Two common causes for friction are a failure to replicate user specific circumstances that cause buggy behavior and incompatible software releases that break critical functionality. Existing test generation techniques are insufficient. They fail to test candidate patches for post-deployment bugs and to test whether the new release adversely effects customer workloads. With existing test generation and deployment techniques, users can’t choose (nor validate) compatible portions of new versions and retain their previous version’s functionality.

We present two new technologies to alleviate this friction. First, Test Generation for Ad Hoc Circumstances transforms buggy executions into test cases. Second, Binary Patch Decomposition allows users to select the compatible pieces...
of update releases. By sharing specific context around buggy behavior and developers can create specific test cases that demonstrate if their fixes are appropriate. When fixes are distributed by including extra context users can incorporate only updates that guarantee compatibility between buggy and fixed versions.

We use change analysis in combination with binary rewriting to transform the old executable and buggy execution into a test case including the developer’s prospective changes that let us generate and run targeted tests for the candidate patch. We also provide analogous support to users, to selectively validate and patch their production environments with only the desired bug-fixes from new version releases.

This paper presents a new patching workflow that allows developers to validate prospective patches and users to select which updates they would like to apply, along with two new technologies that make it possible. We demonstrate our technique constructs tests cases more effectively and more efficiently than traditional test case generation on a collection of real world bugs compared to traditional test generation techniques, and provides the ability for flexible updates in real world scenarios.

**Keywords:** test generation, change analysis, binary analysis, binary rewriting

1. Introduction

Developer testing may not be representative of how software is used in the field [1] and many test suites are insufficient [2]. User bug reports [3, 4] and vulnerability disclosures [5, 6] are populated primarily with bugs discovered in the field when users or third-party security analysts use the software in ways that the developers had not tested before deployment. User bug reports typically include some evidence of the bug, such as memory dumps, stack traces, system logs, error messages, screenshots, and so on, but are often insufficient for the developers to reproduce the bug [7]. Even vulnerability disclosures are sometimes incomplete, making it difficult for developers to reproduce the reported exploits [8]. Thus when a security vulnerability or other critical bug is not
detected by developer testing prior to deployment, but reported by users, developers need to construct a new test that both reproduces the bug in the original version of the code and verifies the absence of the bug in the patched code. Aside from patching, deployment presents another issue since software updates may not be compatible with existing infrastructure in some user environments. As a result, customer organizations may avoid updating their installations, leaving buggy code in production.

Enterprise software update procedures revolve around an interaction between user and development organizations shown in Figure 1. This interaction gets increasingly complex as a single software provider services hundreds of different user organizations. The developer organization writes code that gets checked into a version control system (VCS) and built with continuous deployment/continuous integration (CI/CD) which ships the resulting executables to user organizations that deploy the software. Software operators (IT staff) within
the user or customer organization approve the update and install the new executable on machines.

Awkward interactions between the user organization and the development organization often cause the traditional software update model to fail leading to insecure or nonfunctional installations. When users report bugs, developers need to reproduce the buggy behavior in the developer environment, update their test suites, and develop a prospective patch. The current update paradigm may fail to incorporate information from the specific user instance that caused the buggy behavior relying solely on bug reports to assist developers. This means a developer must manually build a representative test case that reproduces the bug in the original code and verifies the absence of the bug in the patched code.

Every change has the potential to include unwanted side effects and while CI and CD provide some protection it only considers the perspective of the developer organization. In the event of a problematic update, the operators responsible for deployment have no recourse other than to submit new bug reports. This interaction gets further complicated by the fact that most bug fixes are incorporated as part of more general releases which include changes other than the bug fix. These additional changes may in fact break existing functionality on any given installation even if they pass tests during CI/CD.

Interaction during reporting and distribution fail for the same reason: restricted context. The user organization will not have access to the source code producing the software and cannot make tailored modifications, and the developer organization must support many different user organizations without access to any specific installation. This handshake between parties demands operators and developers have intimate knowledge of what the other organization needs while also inherently preventing them from sharing information.

We propose a new update paradigm that exposes precisely the relevant information needed by both organizations that supports the interaction between these groups while still keeping their roles distinctly separate without imposing prohibitive overhead. We give an overview in the next section.
2. Technical Innovations

Our new patching workflow relies on two technical innovations, 1) Test Generation for Ad Hoc Circumstances which we implement in a prototype called ATTUNE (Ad hoc Test generation ThroUgh biNary rEwriting), and 2) Binary Patch Decomposition (BPD) which provides granular control over traditional software updates which is included as part of ATTUNE.

Figure 2 shows the gist of our new bug fixing workflow. Let’s consider a single bug as an example. First the bug is identified in the production environment. A lightweight system, like [9], records the production software without introducing significant overhead and produces a log that can be replayed later. Lightweight recording techniques may not capture enough information to faithfully reproduce some bugs. In that case, we could substitute Zuo et al.’s new approach [10], which is very lightweight but requires a series of recordings where the same bug reoccurs to eventually capture enough data for faithful re-
production. This lightweight recording captures all sources of non-determinism required to recreate the buggy execution. Then the log is augmented with additional information with an offline heavyweight recording process that includes required additional information as outlined in 3.1. The verbose log shares appropriate context between customer organization software operators (IT staff) and developers maintaining the code. With this augmented log the developers can re-create the bug from the production environment. Using our novel test generation technique for ad hoc circumstances (Ad Hoc Test Generation), outlined in 3.2 and 3.3, developers turn this augmented log into a repeatable test case for prospective patches. That test case becomes part of the test suite and the developer approved patch gets added to the version control system (VCS) along with any other changes as part of standard development. In the event multiple users report the same bug, there may be redundant test cases. The VCS still integrates with continuous integration and continuous deployment systems (CI/CD), but in order to expose additional context to the operator it interacts with our BPD changelog datastore. Should an update received from developers break critical functionality and fail manual approval, an operator has the opportunity to leverage the context in the BPD datastore to craft a custom partial update for their customer organization.

By sharing specific context in the verbose trace and the BPD datastore between parties the developers automatically have a test case to fix bugs and operators deploying software have the flexibility to build updates that meet their needs. Developer practices limit the level of granularity in the current prototype of the BPD datastore so tangled commits (single commits with multiple unrelated or weakly related changes [11]) create some confusion. With additional effort from the developing party or minor changes to the BPD datastore prototype of course these commits can be untangled.

The first of the two technologies that make this possible is test generation for ad hoc circumstances which we call ad hoc test generation because the generated test emulates the ad hoc user context that manifested the bug. The key observation that makes ad hoc test generation plausible is analogous to Tucek
et al.'s “delta execution” [12], whose large-scale study of patch size found that
security and other patches solely to fix bugs tend to be modest in size and
scope, rarely change core program semantics, shared memory layout or pro-
cess/thread layout. Nonetheless, bug reproduction is difficult. The premise of
record-replay technology highlights the difficulty in capturing all of the con-
ditions that led to erroneous behavior and recreating those conditions in the
developer environment. Standard bug reports often fail to capture all the rele-
vant state information, and this paper addresses the feasibility of using ad hoc
test generation in such scenarios. We have developed ATTUNE as a solution
that combines the buggy executable with the modified version to emulate what
would have happened had the modified version been deployed during the buggy
execution.

Instead of requiring developers to build doubles, mocks or other test scaf-
folding to fake the user environment for its tests, ATTUNE builds on existing
record-replay tools. It takes all non-deterministic inputs and replays them as
they happened when the bug manifested. This eliminates the common com-
mutation failures when a developer tries to recreate the execution from user
reports. Furthermore, since all non-deterministic inputs are replayed Ad Hoc
Testing eliminates the possibility of confusing flaky tests [13].

There are two main challenges to technically implementing Ad Hoc testing.
1) How do you accurately identify changed portions of the executable once the
source code level abstractions have been stripped away? and 2) How do you
replay events after executions have diverged?

ATTUNE solves these challenges with two key insights: 1) The identifiers
used in the source code rarely change, and are still represented in the executable.
Software patches rarely modify function names and global variable names. In the
event they are changed, a mapping exists between the old and new identifiers so
these points still provide consistency between the old version and new versions.
These locations provide landmarks amidst the unstructured binary data to guide
ATTUNE’s manipulation. 2) The recorded log does not need to be replayed
verbatim in order. Events in the log can be skipped or swapped, and new events
can be derived from those in the log, to match the patch. Our runtime emulation
algorithm selects which events to replay and when to support execution after
divergence.

In addition to Ad Hoc Test Generation we offer binary patch decomposition
to support deployment of software that would otherwise be impossible. Cus-
tomer organizations tend to update software only when necessary for fear of
updates introducing side effects that disrupt service. Software releases accord-
ing to a common schedule, often contain many modifications most of which a
singular user would deem unnecessary. These pose an unnecessary risk and may
not be able to integrate new versions if relevant interfaces have been replaced
or wiped away. This leaves many users in an awkward position: they have code
with known deficiencies and the corresponding updates, but they also can’t
apply those updates. As an example Equifax’s well known breach in 2017 [14]
exposed in 145.5 million people even though a bug fix was available

Figure 3 outlines our solution to this problem. The customer organization
software operators (IT staff) monitor the deployed software and submits the
bug report to the developer. While the developer works, the binary patch
decomposition datastore (detailed in Figure 13) runs incremental builds and
tracks changes at the binary level. When the developer distributes the new
release, operators can instead apply partial updates from the BPD datastore
if the new release is incompatible with existing infrastructure. Ad hoc test

generation allows the operators to test the partial updates on recorded workloads
to verify the partially updated software functions correctly.

An earlier version of this work [15] introduced ad hoc test generation, and
briefly discussed our technique for adding developer environment metadata to
patch releases, enabling operators to validate patched versions with their own
workloads. In this paper, we build on our previous ad hoc test generation
workflow to enable a more complete solution. Furthermore, we added new
functionality to allow for partial updates, e.g., when a full update would break
mission critical functionality, based on ideas we previously sketched in [16].

The new contributions of this expanded paper are:

• A method for decomposing full version updates, with multiple bug-fixes
  (and possibly new features), into its component pieces to enable partial
  updates.

• A testing framework for determining if a partial update is compatible with
  existing user environment infrastructure.

• A patching technique that allows users to apply partial updates despite
  not having access to the source code.

We explain our requirements for verbose execution traces and the technical
details of our binary rewriting techniques in Section 3. Our evaluation in
Section 4 describes how a developer would use ATTUNE to test candidate
patches for a variety of CVE security vulnerabilities and other bugs from well-
known open-source projects. Section 4 also gives an example where the user
records their own workload with the original program and replays with the
modified program to convince themselves that the bug has been fixed and the
patch does not break other behavior. We also test our partial patching in-
frastructure applied in the user environment to show partial updates can in
fact fix the bug. We analyze threats to validity in section 4.4 and compare
to related work in Section 5, and then summarize this work. Our open-source
3. Implementation

Our ad hoc test generation workflow has four main components: recording, static preprocessing, load-time quilting and the runtime replay decisions. We detail support for Ad Hoc Test generation in the customer environment when source code is unavailable in section 3.5. Recording and the two preparation stages are shown in Figure 4, with runtime depicted in Figure 12. Both preparation stages leverage the open-source Egalito recompilation framework [17].

3.1. Recording

We assume production recording with the user’s choice of lightweight tool and, when warranted by some external mechanism that detects an error or exploit, offline replaying that tool’s recording while re-recording with rr’s recorder [18] as in Figure 2. Instead of rr, any other recording engine that constructs
sufficiently verbose traces would suffice, but we do not know of any actively-
supported open-source alternatives. Specifically, the trace must provide the
details needed for ATTUNE to recreate the successive register contents and
memory layouts leading up to when the bug manifested. Thus the recorded
sequence of events must include register values before and after system calls,
files that are mapped into memory, and points at which thread interleaving and
signal delivery occur during execution.

ATTUNE imposes almost no restrictions on the user after constructing the
verbose trace. ATTUNE’s technical design decisions enable ATTUNE to run
without privileges in user-space, with conventional hardware, operating system,
compiler, libraries, build processes etc. and no changes to the application or the
accompanying libraries. This represents a stark contrast to other test generation
techniques like symbolic or concolic execution. While the technical details of
our binary rewriting mechanisms are specific to our implementation ad hoc test
generation is not, and in principle ATTUNE prototypes could be built with any
record-replay technology that supports sufficiently detailed execution traces on
any architecture.

There are some changes that ATTUNE does not support. While our tech-
nique will capture concurrency related bugs, it cannot verify patches since it is
impossible to verify what nondeterministic thread interleavings might do after
a change. Any significant changes to data structures e.g., changing the size of a
struct on the stack or in the heap, that would require changes to memory allo-
cation would not be tolerated. Any changes to preprocessor macros don’t have
symbols associated with them and so are not supported. Of course any major
feature addition that fundamentally changes software behavior is not supported.
We support all other changes that can be associated with symbols in the binary.

3.2. Static Preprocessing

Source Code and Binary Preprocessing. Figure 5 shows an abbreviated
element patch file from a libpng bug-fix [19]. Patch files document which files
changed, which function in the file changed, and which lines within that function
--- a/pngrutil.c // file info
+++ b/pngrutil.c
@@ -3167,10 +3167,13 @@ png_check_chunk_length(...)
 { ...
 - (png_ptr->width * png_ptr->channels
+ (size_t)png_ptr->width
+ * (size_t)png_ptr->channels

 Figure 5: libpng-1 Abbreviated Example Patch file
 were inserted and deleted. Patch files are created with a standard format so we
 are not limited to a single diff implementation.

 Dwarf Information & Symbol Table. Patch files don’t provide any
 information about the resulting binary. Since the recorded trace relies on bi-
nary/OS level information (register values, pointers, file descriptors, thread ids,
 etc.), we need to translate from changes in the source to changes in the binary.

 182: 0000000000003fe0 56 FUNC
   GLOBAL DEFAULT 1 png_check_chunk_name
 183: 0000000000004020 221 FUNC
   GLOBAL DEFAULT 1 png_check_chunk_length
 184: 0000000000004100 172 FUNC
   GLOBAL DEFAULT 1 png_read_chunk_header

 Figure 6: libpng-1 Symbol Table Entries

 ...
 <> DW_AT_producer: (indirect string, offset: 0x1d90): GNU C11 7.4.0 ...
 <10> DW_AT_language 12 (ANSI C99)
 <11> DW_AT_name: (indirect string, offset: 0x1c8e): pngrutil.c

 ... 0x0000402b [3156, 0] RS
 0x0000403a [3166, 0] RS
 0x00004046 [3182, 0] RS

 Figure 7: libpng-1 Relevant DWARF Line Entries

 Two mechanisms enable this translation: The first is the symbol table stan-
The key insight is that the symbols act as a point of reference between the old and the modified binaries. They remain unchanged even if their addresses and references change. After processing the patch file, we use the symbol tables to find the locations of functions and global variables, and we use DWARF information for finding changed lines and identifying source files. These two sources combined contain all the information in the source level diff at the binary level. Refer to Figures 6 and 7 for concrete examples.

Most real-world builds create multiple binaries and associated libraries, so it may be unclear which binary contains the associated change. In order to generalize to sophisticated build processes, ATTUNE uses DWARF information to search through all re-compiled binaries to find the modified file.

3.3. Load Time Quilting

Pre-Load Steps for Quilting. Once the function and line addresses have been resolved, and a prospective patched binary has been compiled, we can generate our test code. In order for the newly compiled patched code to remain a viable test case, it must maintain the binary context of the original code. While most of the binary context remains unchanged, code pointers and data pointers that point somewhere inside the modified functions or that point from the modified functions to any location outside of the modified binary must be updated.
accordingly. To create the most accurate test we point to the original binary context wherever possible. In order to fully integrate the patched code with the recording, references to shared libraries must point to where the shared libraries were loaded in the recording, references to places in the modified section of the code must point to the appropriate place in the patched code, and references to unmodified contents of the patched binary must point to the appropriate place in the original binary as in Figure 8.

In order to prepare for load time quilting resolution (explained shortly), static reference identification needs to occur for bookkeeping purposes. The patched function is scanned for all symbol references that need to be resolved to integrate with the recorded context. Some references like references to locations within the modified function (such as jump and conditional jump instructions) can remain unaltered in position independent code. So after all references are accounted for, they are trimmed to the subset of references that need to be changed during the quilting procedure. This includes references to strings, shared library functions, functions that only exist in either the original or the modified binary, functions that exist in both, procedure linkage table (PLT) entries, and global variables. Since symbols are the points of reference between original and patched binaries, because recompilation renders addresses meaningless, references to be resolved are defined as a symbol and an offset from that symbol.

**Loading Replication & Custom Loading.** In modern Linux systems the system loader is responsible for parsing the executable's header, loading it into memory, and dynamic linking. Since shared libraries are not always loaded at the same positions, references related to the global offset table (GOT), and procedure linkage table (PLT) are resolved after loading completes. Even though ATTUNE knows pre-load which references need resolution, it can't actually resolve those references until load time. To preserve the integrity of the replay, all required shared libraries, executables, and system libraries must be loaded into the recorded memory locations. The trace includes shared libraries and executables required for replay, and non-recorded libraries loaded during replay.
are limited to the system loader, which is required at the start of any process.

In order to replicate the recorded loading activity, ATTUNE begins by loading a small entry point program (replay hook) that hijacks execution from the system loader and begins the replay process. Since some references in the patched code can’t be resolved until the original code is loaded into memory, so initially loading replicates exactly what was recorded. Once the original segments are loaded into memory and GOT/PLT relocations are completed, ATTUNE resolves remaining references in the patched code (described below).

Finally, ATTUNE’s loader loads the quilted code after finding an appropriate place to put it. Note quilting has to be repeated on every replay, and the files containing the original and patched executables are not modified. The loader searches the address space for the lowest slot large enough to accommodate all of the patched code, then loads the patch following the Linux loading conventions. Figure 8 depicts the address space when loading has completed, and Algorithm 1 outlines the loading procedure.

Algorithm 1: Custom Loading Algorithm

Result: Load patched code into the address space

```plaintext
code_seg_size = 0; char* code_buf;
for func in mod_funcs do
  code_seg_size += func.size
end
for segment in addr_space do
  space = next_segment.start - segment.end;
  if space > code_seg_size then
    start = segment.end;
    for func in mod_funcs do
      patched_code = func_gen_code;
      code_buf += patched_code;
    end
  end
end
```

15
Address Translation Procedure. A summary of the procedure to translate pointers from the context of the modified binary to the context of the original binary is given in Figure 9, and consists of both pre-load and load-time actions. The process starts from the address of the modified function as determined from the patch file and DWARF processing. ATTUNE scans the modified function for references. If a reference is affected by the quilting process, then ATTUNE’s translation procedure corrects the pointer.

The log messages in Figure 10 explain the process in detail: An instruction in the patched binary at 0x1b214 points to 0xaa60. In order to update the instruction to point to the same position in the original binary we need to identify the correct symbol and offset in the original. First we convert the target address 0xaa60 into a symbol and offset in the patched binary. Since this instruction is just calling a function, the target symbol is the function name and the target offset is 0. Then ATTUNE searches the original binary for the same symbol and offset, and in this case the function was generated at the same address in original binary. Resolving string references, global variable references, and PLT references require slightly different procedures and are described below.

Finally the patched code is generated with instructions pointing to the correct locations at runtime.

PIC Code, PLT Entries & Trampolines. Position independent code compilation has become the standard for security and efficiency reasons, so
Linking function: png_check_chunk_length
in module pngutil
Updating Instruction Reference
from [0x1b214] to [0xaa60]

//identifying reference point
Target Symbol: png_chunk_error
Offset From Symbol: 0
Symbol Location in original binary:
0xaa60

//target address in the original binary
Target Address: 0xaa60
...
//patch references string
Resolving string reference at: 0x1b2cd
Resolving offset ... for "chunk data is too large"
//identified string in original binary
Found string: "chunk data is too large"
  at 0x320e
  ... module pngutil code found at 0x000000
  ... module pngutil data found at 0x200000
  ... generating quilted code

Figure 10: libpng-1 abbreviated linking example
modern binaries can be loaded anywhere in the address space. As a result the locations of external functions and symbols are not known until those symbols actually exist in the address space. Since most library functions aren’t called, they aren’t all resolved at load time and instead are resolved only after they are called. The procedure linkage table (PLT) acts as a table of tiny functions that perform a function lookup and trampoline to where the code for external functions are defined.

Unfortunately, we can’t rely on a PLT because the system loader that performs the runtime function resolution doesn’t know about ATTUNE’s special memory configuration. Two key differences let us implement static trampolines instead of relying on the traditional PLT mechanism. 1) We only need to resolve the PLT entries that are referenced by the modified code, which comprise a small fraction of the overall PLT, and 2) we can resolve these beforehand without relying on the PLT’s lazy loading mechanism because the shared libraries have already been loaded by the time this code is injected. The x86.64 architecture only allows call instructions with a 32-bit offset, but we need to call functions across the 64-bit address space to reference shared library functions. To accomplish this we transform calls to PLT entries into a move instruction that loads an address into a register, and then a call instruction to the address in the register, as shown in Figure 11.

**Resolving String & Data Sections.** The patched code may also reference data section variables like global data and strings. The patched code must reference the old code where possible and the patched code where required. Identical symbols and strings function act as points of reference between the modified and the original binary.
These translations are similar to Figure 9, with a few minor differences:

String tables don’t have an associated symbol table. The modified code references the string directly, but to lookup the location of a specific string in the original, we have to iterate through all of the read-only data. If the string exists in the original binary, then we point at it, otherwise ATTUNE points to the appropriate location in the new data section. Note the binary normally accesses data through a global offset table entry, but cannot use it here because the global offset table was compiled for the modified code. Instead, ATTUNE transforms the binary to point to the data directly, since it knows where the data has been loaded.

3.4. Runtime Replay Decisions

The runtime architecture is shown in Figure 12. At runtime we continue to leverage developer environment information to aid ATTUNE’s decision making, e.g., we know exactly which functions have been modified and perform a strict replay until a modified function is called. We break at that point and move to the patched code, where we use information about added or deleted lines to inform decision making.

For any non-deterministic event that takes place during replay, we must decide whether to use a corresponding event recorded in the log or to actually
submit the event for operation by the kernel, i.e., execute live as would be
required if the inserted code makes a new system call. We emulate kernel state
and kernel events whenever possible, and only ask the kernel to perform the
replaying action when necessary, following the greedy approach shown by the
pseudocode in Algorithm 2. It should be noted that system calls which depend
on process state, like malloc, and mmap, don’t require emulation since this
state is actually recreated during replay. All file operations performed during
replay are based on information available from the recorded trace, essentially
recreating how the program would have acted at the time of the bug except
now (for successful patches) without the bug. If there is no further information
available, the emulation ends.

**System Calls.** The simplest event types to replay are system calls that
don’t involve file IO. We can reuse results from the log if the parameters for
the syscall match what is in the log. It won’t match the log exactly since the
log contains checks for all registers including the instruction pointer that is
obviously different, but we relax these checks once replay has diverged to only
check registers containing syscall parameters.

**File IO.** System calls involving file, network or device IO are harder to
replay since they require a specific kernel state. We have to recreate the file
state so we track open, close, stat, read, write, and seek operations for all
file descriptors during replay. At the point the replay diverges we have a partial
view of the file system. Of course we can’t recreate any data that doesn’t exist,
but if a file operation can’t be satisfied during replay we can look forward in the
recorded trace to see if we have enough information to satisfy the operation. If
we do then we emulate it, and unfortunately if we don’t we have to die. Another
approach would be to supply random bytes, but we feel this wouldn’t accurately
reflect a realistic state if the full file system were available.

**Signal Delivery.** If a signal is intercepted by the emulation engine, we
need to decide if that signal should be delivered to the replaying process. Our
normal replay mechanism based on rr’s replay mechanism determines signal de-
elivery based on the value of the *retired conditional branches* (RCB) performance
counter standard in Intel chips. For signals that have been recorded, we check if we are in an inserted line. If we are then we deliver the signal and assume it is created by the patch (such as a segfault from an incorrect memory reference in the patch). However, if a recorded signal is delivered and we are not currently in the inserted section of the code we can do our best to estimate at what RCB count it should be delivered by taking the target RCB count and adding the number of RCB’s caused by inserted lines. While this isn’t perfect it does allow for a rough idea as to when the signal should be delivered. In the event an unrecorded signal fires we allow that signal to be delivered without interference since there is no recorded timing information to guide delivery.

Algorithm 2: Runtime Replay Algorithm

Input: e: an event that stops replay

Output: The next event to replay

Function getResult(e):

if !diverged then
    return next_recorded_result;
if is_syscall_without_file_io && exists_unused_in_log then
    return recorded_result;
if is_syscall_with_file_io && supported_operation && exists_unused_in_log then
    return recorded_result;
if is_signal && signal_is_recorded then
    if current_pos == inserted_code then
        return nullptr; // execute live
    return DELAY; // delay until RCB count
return nullptr; //execute live

3.5. Binary Patch Decomposition and Partial Updates

Since all the information for ad hoc testing is at the binary level at runtime, ATTUNE supports software operators (customer organization IT staff) who don’t have source code but still want test potential updates with their own specific workloads before deployment. The only requirement is that the developers
are willing to include metadata describing where the changes in the executable took place and what they consisted of. To support the developer distributing this information we developed a novel technique we call binary patch decomposition (BPD), which integrates with the existing build process. In simple terms, BPD breaks a full update down into its component pieces and their contents. Its complexity lies in tracking dependencies between updates such that the version of the software in the test has the proper contents.

Figure 13 outlines the data structure that makes this possible. We integrate with the version control system as the software is developed and track which modifications are associated with each commit. In detail, since our ad hoc testing generation technique depends on symbols in the binary we construct the metadata that allows the operator to apply the patch based on the symbols that change. We also track the symbols each piece of code depends on and the associated versions. Along with the symbols and versions we also store the contents of those symbols to apply when the ad hoc test is run.

Based on the contents of the datastore, the developers can release the binary specific metadata which details the changes that comprise the update. The algorithm for constructing the partial update metadata is described in Algorithm
3. The algorithm takes the original and new binaries as inputs, and then for
every changed symbol it has to do two things. First, if the symbol exists in the
old binary, it must add the old size and position data so any references to this
piece can be removed. Second, BPD must search through all the dependencies
of each changed symbol (a dependency is any nonlocal reference); if the depen-
dency existed in the old binary (as per the symbol name), then the metadata
can simply add the new code piece and the location of its own dependencies.
If the dependency doesn’t exist, it must be added to the metadata as a newly
changed symbol and its dependencies searched as well.

The software operator can check the update for compatibility, and in the
event a full update is incompatible with existing infrastructure they can apply
a partial update that may still support the old infrastructure. That partial
update may consist of as many individual patches as they would like to include.
In the event that selected patches are incompatible (including multiple versions
of the same symbol) the newest version of the symbol is used.

Unlike ad hoc test generation in the developer environment the operator
needs to export the test to a modified executable which can be deployed. This
distinctly differs from run time quilting as the requirements to keep memory
layout the same no longer make sense. Leveraging Egalito’s binary rewriting
capabilities we completely remove the modified symbols, and replace them with
the correct versions. Since Egalito provides arbitrary rewriting, we do so with-
out leaving behind any software bloat or extra instructions that would impose
performance problems. Effectively we are recompiling the binary to some in-
termediate build between version releases despite not having the source code.
ATTUNE’s binary rewriting technique avoids the need for recompilation mak-
ing it more efficient by both saving compute cycles, and eliminating the need to
store source code changes in the BPD datastore. Furthermore by not recom pil-
ing from selected source code snippets developer practices remain uninterrupted
without exposing any source code to the operators (customer organization IT
staff) deploying the software.

As currently constructed ATTUNE requires the developers to ship the entire
BPD datastore to customer organizations, but in a commercial setting this datastore would be made available through a shared resource as depicted in Figure 3.

Since our BPD technique integrates with git to perform intermediate builds, BPD’s ability to create the minimal update is somewhat limited by developer practices. The following circumstances merit further discussion: 1) In the simple case when one bug-fix (involving an arbitrary number of functions) corresponds with one commit, BPD handles this easily. 2) When multiple bug fixes are intertwined in the same commit, git does not provide any way for the developer to distinguish which functions/symbols are associated with each bug-fix so BPD requires that developers update the datastore appropriately. 3) If the same function is updated as part of multiple bug-fixes in different commits, the BPD datastore provides operators access to each version of that function. 4) If the same function is changed as part of multiple bug-fixes in the same commit then the BPD datastore does not support this because there are no intermediate builds per bug-fix to extract different versions of the same function. 5) If a bug fix involves thread interleavings while ATTUNE’s ad hoc test generation provides no guarantees, binary patch decomposition supports apply selected patches of this nature.

4. Evaluation

We evaluated ATTUNE on a Dell OptiPlex 7040 with Intel core i7-6700 CPU at 3.4GHz with 32GB memory, running Ubuntu 18.04 64bit, using gcc/g++ version 7.4.0 and python 3.4.7. ATTUNE is built using CMake version 3.10.2 and Make version 4.1.

Since we want to evaluate ATTUNE on an unbiased selection of patches for both security vulnerabilities (CVEs) and other kinds of bugs, and know of no benchmark that provides user environment execution traces or scripts to set up the user context for recording traces, we recruited (for one semester of academic credit) an independent challenge team of three graduate students.
Algorithm 3: Pseudocode: Metadata Construction

Result: Metadata(old_info, new_info)

Input: parsed original binary OV; parsed new binary NV; BPD datastore DB

Output: metadata information required to construct patch MD

getMetadata (OV, NV, DB)

- original_code = DB.get_code_pieces(OV);
- new_code = DB.get_code_pieces(NV);
- res.new_info ← ∅, res.old_info ← ∅;
- changed_symbols = DB.getChangedSymbols(NV);

foreach symbol ∈ changed_symbols do
    res.new_info.add(symbol);
    if symbol ∈ original_code then
        res.old_info.add(symbol);
        foreach cp ∈ symbol.dependency_list do
            if cp ∉ original_code then
                sym = Symbol(cp, newChange=True);
                res.new_code.add(sym);
        end
    end
end

return Metadata(res.new_info, res.old_info)
who were not involved in developing ATTUNE nor versed in how it works. They were tasked to identify a diverse collection of around 20 bugs in widely used C/Linux programs. The bugs had to have been patched during 2016–2019 and the students had to construct user contexts that demonstrated the buggy behavior. For example, in order to recreate the circumstances leading up to the redis-1 bug, first one needs to run the server with a specific configuration, connect to the server in MONITOR mode, and then send a specific byte stream to the server. Note the team could script creation of such contexts given the bug and its root cause is already known; record/replay is for capturing and reproducing the contexts of previously unknown bugs. The team identified the 21 bugs listed in Table 4.

4.1. ATTUNE successfully validates a wide range of patches provided that corresponding metadata is available

ATTUNE successfully validated the real developer patches in both the developer and operator environments for 19 and failed for 2 of the bugs the challenge team collected, marked with ✓ and ✓ in Table 4 resp. We organize the 19 bugs successfully handled into several different types and describe how the developer employs ATTUNE in each case, then explain the 2 failures.

String Parsing bugs are fairly common as there are often many corner cases, which can have significant security implications since input strings may act as attack vectors. Figure 14 [20] adjusts Curl’s treatment of URLs that end in a single colon. In the buggy version, Curl incorrectly throws an error and never initiates a valid http request. The patch modifies one file. Since ATTUNE replaces the entire modified function instead of individual lines of code, it needs to resolve all references in the new version.

ATTUNE uses the recorded execution to recreate the context that triggered the bug, and then jumps to the patched code upon entering the modified function. Since the only change was adding an if statement that doesn’t trigger a recorded event, the ad hoc test continues past the point where the bug occurred, without divergence other than instruction pointer and base pointer. The devel-
...+
+ if(!portptr[1]) {
+     *portptr = '\0';
+     return CURLUE_OK;
+ }
- if(rest != &portptr[1]) { ...;
-      ...
+     *portptr++ = '\0'; /* cut off the name there */
+     *rest = 0;
+     msnprintf(portbuf, sizeof(portbuf), "%ld", port);
+     u->portnum = port;
...  

Figure 14: Curl-1 URL Parsing

oper can set a breakpoint at the patched section, watch the if statement process
the input correctly and verify the string in *portptr. The test then ends since
the log has no information regarding how the network would have responded to
the http request had it been sent.

Figure 15 [21] deals with mishandling URL strings crafted with special char-
acters, e.g., the "#@" in http://example.com#@evil.com caused Curl to er-
ronously send a request to a malicious URL. The patch calls sscanf with a
different filter string. Since the surrounding function handles all the URL pars-
ing for the application, it is rather large with lots of references. Unlike the
above bug, which only requires resolving pointers to old strings, the new filter
string needs to be loaded into a new data section and referenced appropriately.

ATTUNE recreates the state that caused the initial behavior and then jumps
to the modified code. There the developer can verify the patch by checking the
values in protobuf and slashbuf.

Mathematical Errors can have security implications when related to pointer
ers or integer overflows. For example, a malicious PNG image triggers a bad
calculation of row_factor in Figure 16 [19], causing a divide-by-zero error and
Denial-of-Service (DoS). With traditional bug reports, the user would need to
send the image as an attachment, but a legitimate user affected by the DoS is
static CURLcode parseurlandfillconn(...) {
    path[0]=0;
    rc = sscanf(data->change.url,
            "%15[\n:\]%3[/\]%[\n]/%[^\n]",
            protobuf, slashbuf, conn->host.name, path);
    if(2 == rc) {
        ....
    }
}

Figure 15: Curl-12 String Parsing

png_check_chunk_length(...) {
    ...
    size_t row_factor =
        (png_ptr->width * png_ptr->channels
        + (png_ptr->bit_depth > 8? 2: 1)
        + 1 + (png_ptr->interlaced? 6: 0));
        (size_t)png_ptr->width
        + (size_t)png_ptr->channels
        + (png_ptr->bit_depth > 8? 2: 1)
        + 1
        + (png_ptr->interlaced? 6: 0);
}

Figure 16: libpng-1 Mathematical Error
unlikely to be aware of the carefully crafted malicious image uploaded by an attacker. ATTUNE does not require attachments besides the execution trace, since the re-recorded trace includes the image. After the developer writes the patch, they use ATTUNE to verify that row_factor is no longer 0. The patch doesn’t trigger any new events so the function returns gracefully.

**New Functions & Function Parameter Refactoring.** Many fixes, especially those that pertain to size miscalculations, involve refactoring the buggy function to require a new parameter or writing an entirely new function (with new DWARF and ELF metadata). While not particularly strenuous for the developer, these types of fixes create a challenge for ATTUNE. Since both the function that has been refactored or inserted and the functions that call the new/refactored function need to be modified, ATTUNE must replace all these functions in the executable and properly link them.

A patch for the `wc` file processing utility adds special character parsing...
void addReplyErrorLength
    (client *c, const char *s ...)
{
- if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)) {
+ if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE) && !(c->flags & CLIENT_MONITOR)) {
+     char* to = c->flags &
+     CLIENT_MASTER? "master": "replica";
... 

Figure 18: redis-1 Erroneous Conditional

url_parse (const char *url ...)
{
    ...
    + /* check for invalid control characters in host name */
    + for (p = u->host; *p; p++) {
    +     if (c_iscntrl(*p)) {
    +         url_free(u);
    +         error_code = PE_INVALID_HOST_NAME;
    +         goto error;
    +     }
    + }
    + }

Figure 19: wget-2 New Loop
functions as shown in Figure 17 [22]. ATTUNE loads patched versions of the
new function and those functions that call the new function into the address
space. The new function is loaded to point towards the original libraries and
executables where appropriate, and the modified calling functions point to the
new function. There is no need to send a file with the problematic non-standard
characters in the bug report to the developer, since it is included in the recorded
log. These types of bugs can be difficult for conventional bug reports as files in
transit may arrive with modified encoding types and changed contents.

ATTUNE provides the input from the recorded file and successfully returns
from the modified functions displaying the patched output. Testing the modi-

fied `wc` code doesn’t diverge drastically from the original execution trace. The
developer can verify the patch by letting the program run to termination and
inspecting the calculated value.

Adding Conditionals. Perhaps the most common patch we saw involved
adding conditionals. Many security-critical patches make one-line changes to
correct conditional checks. We examined one such example in `redis`. Such
services are particularly hard to test and debug using conventional mocks, as
complex network inputs can be difficult to recreate in mocking frameworks. Re-
dis allows monitor connections to send logging and status checking commands.
The buggy version in Figure 18 [23] didn’t check the client flags for the monitor,
which resulted in a kernel panic. While this was one of the smaller patches, the
validation process varied substantially from the log. ATTUNE enables the de-
developer to step through the program and watch progress through the modified
control flow past the point of the crash.

New or Changing Loop Conditions. Bad loop conditionals are also
common. Reference resolution is performed as before, but these patches vary
greatly from an ad hoc testing perspective because loop conditionals do not
necessarily exhibit the bug on the loop’s first iteration. One such example from
the `wget` utility demonstrates how ATTUNE handles this sort of change in a
security-critical situation. The bug allowed attackers to inject arbitrary HTTP
headers via CRLF sequences into the URL’s host subcomponent. Attackers
could insert arbitrary cookies and other header info, perhaps granting access
to unauthorized resources. The developer modified the url_parse functions in
Figure 19 [24] to check each character in the host name and throw an appropriate
error. During ad hoc testing the developer verifies the patch works by watching
the program check each character, and upon entering the if statement freeing
the URL pointer and proceeding correctly to the error handling code.

**Swapped Code:** ATTUNE successfully constructed test cases in scenarios
that swapped library function calls yes-1 [25] and swapped control flow blocks
df-1 [26]. The yes-1 patch makes far-reaching changes across the code base to
address the same bug in multiple places (15 files). Assuming the recorded log
only manifests one instance of the bug, then the generated ad hoc test case can
only check for that instance, not changes elsewhere in the code base.

**Failures:** ATTUNE successfully generated ad hoc test cases for those chal-
lenging patches where the compiled binaries included complete metadata. How-
ever, it failed on functions with no ELF symbol table entry: We were
initially surprised that a removed break statement in shred-1 [27] caused an
error, since the change is so small. Upon investigation, we found this behavior
should be expected, since the function (used only in one place) was inlined by
the compiler – thus no symbol table entry for cross-referencing the function.
ATTUNE also failed due to **DWARF omissions:** Applying ATTUNE to pa-
rameter changes in curl-8 [28] was unsuccessful. We expected to be able to locate
the modified function in the loaded binaries to link the patch, but the DWARF
metadata generated by the compiler did not include the filename for the file
containing that function. ATTUNE depends on the compiler’s compliance with
the DWARF specification.

4.2. ATTUNE’s wait time and memory overhead is small

To get some perspective of ATTUNE’s overhead, we compared ATTUNE
with KLEE [29, 30], a state of the art test suite generation tool. We used
KLEE version 2.1 [31] compiled with LLVM 9.0.1. We limited this comparison
to those bugs in Table 2 from CoreUtils, since KLEE supports CoreUtils eas-
ily. The other bugs we studied have more external libraries, aside from libc, so
would require additional engineering effort for KLEE to accommodate. KLEE’s
test generation time was budgeted to timeout after 60 minutes, as in [29]. We
omitted a comparison to other testing tools that only detect crashing and per-
formance bugs, like typical AFL-based fuzzers, since most of the bugs we studied
were not crashing bugs and we did not consider performance bugs. We consider
this type of testing to be a completely different testing methodology that is not
comparable.

**Ad Hoc Test Construction Time** ATTUNE’s quilting occurs at load time
so runs when each candidate patch is tested. However, since recording allows
for targeted test construction, almost all the overhead introduced by symbolic
execution searching the program space is removed. As shown in Table 2, our
worst case was just under 4 seconds.

**Memory Footprint:** ATTUNE inserts patched code prior to each test ex-
cution, so it incurs some memory overhead at test time, as shown in Table 2.

4.3. Operators validate released patches with their own workloads and apply par-
tial updates if necessary

In the last (optional) stage of the patching workflow, the operator validates
the patch in their own environment to verify no needed functionality has bro-
ken. We integrated our binary patch decomposition datastore so ATTUNE
produces correctly formatted metadata enables operators to select individual
bug-fix patches from new releases containing other unrelated changes. Since
ATTUNE operates entirely in user-space, without the support of hardware,
operating system, and so on, it can run in both developer and operator envi-
ronments. ATTUNE summarizes the “difs” in source and binary code, and

33
<table>
<thead>
<tr>
<th>Bug</th>
<th>ATTUNE</th>
<th>KLEE</th>
<th>ATTUNE Time</th>
<th>KLEE Time</th>
<th>Speedup</th>
<th>ATTUNE Mem</th>
<th>KLEE Mem</th>
<th>Overhead Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc-1 [22]</td>
<td>✓</td>
<td>✓</td>
<td>1.37s</td>
<td>300.046s</td>
<td>99.5%</td>
<td>5.9 KB</td>
<td>108.388 KB</td>
<td>94.5%</td>
</tr>
<tr>
<td>wc-2 [32]</td>
<td>✓</td>
<td>✗</td>
<td>1.277s</td>
<td>na</td>
<td>na</td>
<td>2.8 KB</td>
<td>107.7 KB</td>
<td>97.4%</td>
</tr>
<tr>
<td>yes-1 [25]</td>
<td>✓</td>
<td>✓</td>
<td>3.4s</td>
<td>8.569s</td>
<td>60.3%</td>
<td>10.6 KB</td>
<td>107.09 KB</td>
<td>90.1%</td>
</tr>
<tr>
<td>shred-1 [27]</td>
<td>✗</td>
<td>✗</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>ls-1 [33]</td>
<td>✓</td>
<td>✓</td>
<td>1.6s</td>
<td>19.57s</td>
<td>91.82%</td>
<td>7.4 KB</td>
<td>132.9 KB</td>
<td>94.4%</td>
</tr>
<tr>
<td>mv-1 [34]</td>
<td>✓</td>
<td>✓</td>
<td>3.6s</td>
<td>58.4s</td>
<td>93.84%</td>
<td>4.3 KB</td>
<td>208.2 KB</td>
<td>97%</td>
</tr>
<tr>
<td>df-1 [26]</td>
<td>✓</td>
<td>✓</td>
<td>1.48s</td>
<td>18.869s</td>
<td>92.15%</td>
<td>5.97 KB</td>
<td>151 KB</td>
<td>96.05%</td>
</tr>
<tr>
<td>ls-1 [35]</td>
<td>✓</td>
<td>✗</td>
<td>1.2s</td>
<td>na</td>
<td>na</td>
<td>5.6 KB</td>
<td>113.37 KB</td>
<td>95.06%</td>
</tr>
</tbody>
</table>

Table 2: Comparison to KLEE Test Generation
inserted line addresses:
0x6b
0x6e

deleted line addresses:
0x495AD
0x495B7

patched code:
...
69: jne 0xb9
6b: and 0x2,%eax
6e: lea -0x58090939(%rip),%rdx
75: mov 0x58(%rbx),%rax
...

Figure 20: redis-bug-1 Metadata for User Validation
exports metadata allowing for operator validation and partial updates.

For sample user environment workloads, we used the redis benchmark [36],
which simulates thousands of different requests to the server, and the httpperf
benchmarking tool [37] making thousands of connections. Similar to the re-
dis discussion above, ATTUNE’s validation procedure for the redis patch [36]
utilizes only the metadata it added to the released patch, shown in Figure 20.

ATTUNE needs inserted and deleted line addresses for its runtime deci-
sion algorithm. The metadata’s ”inserted line addresses” and ”deleted line
addresses” are offsets into the relevant files while deleted lines from the original
binary are offsets into the original executable. Inserted lines only appear in the
patch release so their addresses are offsets into the patched codefile that gets
mmapped into memory.

4.4. Threats to Validity
To test our ability to both export and apply partial updates, for each bug
we inspected we exported metadata describing each individual change in the
complete version update. Then we quilted the singular change that fixed the
patch as an operator would apply a single change at a time. Unlike ad hoc test
generation in the developer environment, when the modified executable exists
statically, in the operator environment we provide the ability to export the in
memory ad hoc test to a static file. For every bug in the table the partially
updated patched version successfully fixed the bug.

It’s important to note that the size and scope of the change is not accurately
measured only by the lines of code changed, but also how many references need
to be resolved in the quilting procedure. Table 4 describes the extent of the
changes at the binary level, by tracking how many data reference and code
reference resolutions need to be performed to successfully quilt the patch in.
Table 4 also shows how many individual changes are in each version update.
For each partial update that was applied, the exported version of the binary
successfully fixed the buggy behavior.

**Internal.** As far as we know, no execution traces were recorded when any
of the studied bugs were discovered, so we needed bug-triggering user contexts
that could be recorded. We recorded directly with rr, rather than first using a
lightweight recorder and then re-recording the lightweight replay using rr. Ar-
guably, these user contexts could have been designed to facilitate ATTUNE’s
test generation. This threat is partially mitigated since the carefully crafted
scenarios were developed by three grad students who were not ATTUNE de-
developers and did not know how ATTUNE operates. We did, however, instruct
them on how to use rr. Further, we describe how we imagine a developer would
verify their candidate patches using ATTUNE, but we are not developers on
these projects and lack the developers’ knowledge. This is mitigated to some
extent since ATTUNE generated ad hoc tests for the real developer patches.
Ideally, we would also use ATTUNE to generate ad hoc tests for candidate
patches discarded by the developers, to illustrate how we envision a developer
would leverage ATTUNE to determine that their attempted patch fails to fix
the bug, but we could not find any such commits in the version repositories.
Lastly, since do not have access to production users for any of the programs in
our dataset, we simulated a production workload using a standard redis bench-
mark, which may not be representative of the workloads that production users
would construct to validate the redis patch in their own environment.

**External.** We demonstrate that ATTUNE supports a wide variety of single-
<table>
<thead>
<tr>
<th>Bug</th>
<th>Data Resolutions</th>
<th>Code Resolutions</th>
<th>Buggy Version Tag</th>
<th>Patch Version Tag</th>
<th>Distinct Changes Between Versions</th>
<th>Partial Update Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl-1 [20]</td>
<td>4</td>
<td>31</td>
<td>curl_7.63.0</td>
<td>curl_7.64.0</td>
<td>128</td>
<td>✓</td>
</tr>
<tr>
<td>curl-2 [22]</td>
<td>60</td>
<td>318</td>
<td>curl_7.63.0</td>
<td>curl_7.64.0</td>
<td>128</td>
<td>✓</td>
</tr>
<tr>
<td>curl-3 [28]</td>
<td>6</td>
<td>33</td>
<td>curl_7.33.0</td>
<td>curl_7.34.0</td>
<td>246</td>
<td>✓</td>
</tr>
<tr>
<td>curl-4 [40]</td>
<td>6</td>
<td>71</td>
<td>curl_7.63.0</td>
<td>curl_7.64.0</td>
<td>128</td>
<td>✓</td>
</tr>
<tr>
<td>curl-5 [38]</td>
<td>n/a</td>
<td>n/a</td>
<td>curl_7.61.0</td>
<td>curl_7.60.0</td>
<td>223</td>
<td>✓</td>
</tr>
<tr>
<td>curl-6 [41]</td>
<td>8</td>
<td>26</td>
<td>curl_7.62.0</td>
<td>curl_7.63.0</td>
<td>122</td>
<td>✓</td>
</tr>
<tr>
<td>curl-7 [42]</td>
<td>3</td>
<td>21</td>
<td>curl_7.62.0</td>
<td>curl_7.63.0</td>
<td>122</td>
<td>✓</td>
</tr>
<tr>
<td>curl-8 [43]</td>
<td>273</td>
<td>1032</td>
<td>curl_7.62.0</td>
<td>curl_7.63.0</td>
<td>122</td>
<td>✓</td>
</tr>
<tr>
<td>curl-9 [44]</td>
<td>37</td>
<td>103</td>
<td>curl_7.50.0</td>
<td>curl_7.51.0</td>
<td>333</td>
<td>✓</td>
</tr>
<tr>
<td>libpng-1 [19]</td>
<td>1</td>
<td>6</td>
<td>v1.6.34</td>
<td>v1.6.35</td>
<td>53</td>
<td>✓</td>
</tr>
<tr>
<td>libpng-2 [44]</td>
<td>1</td>
<td>6</td>
<td>v1.6.32beta02</td>
<td>v1.6.31beta02</td>
<td>97</td>
<td>✓</td>
</tr>
<tr>
<td>wc-1 [22]</td>
<td>109</td>
<td>298</td>
<td>v8.30</td>
<td>v8.31</td>
<td>90</td>
<td>✓</td>
</tr>
<tr>
<td>wc-2 [42]</td>
<td>79</td>
<td>155</td>
<td>v8.26</td>
<td>v8.27</td>
<td>69</td>
<td>✓</td>
</tr>
<tr>
<td>yes-1 [25]</td>
<td>234</td>
<td>399</td>
<td>v8.30</td>
<td>v8.31</td>
<td>90</td>
<td>✓</td>
</tr>
<tr>
<td>shred-1 [27]</td>
<td>n/a</td>
<td>n/a</td>
<td>v8.27</td>
<td>v8.28</td>
<td>72</td>
<td>✓</td>
</tr>
<tr>
<td>ls-1 [33]</td>
<td>380</td>
<td>387</td>
<td>v8.29</td>
<td>v8.30</td>
<td>68</td>
<td>✓</td>
</tr>
<tr>
<td>mv-1 [34]</td>
<td>89</td>
<td>204</td>
<td>v8.29</td>
<td>v8.30</td>
<td>68</td>
<td>✓</td>
</tr>
<tr>
<td>df-1 [20]</td>
<td>164</td>
<td>348</td>
<td>v8.28</td>
<td>v8.29</td>
<td>65</td>
<td>✓</td>
</tr>
<tr>
<td>bs-1 [35]</td>
<td>140</td>
<td>296</td>
<td>v8.28</td>
<td>v8.29</td>
<td>65</td>
<td>✓</td>
</tr>
<tr>
<td>wget-1 [45]</td>
<td>8</td>
<td>16</td>
<td>5.0.6</td>
<td>5.0.7</td>
<td>30</td>
<td>✓</td>
</tr>
<tr>
<td>redis-1 [23]</td>
<td>3</td>
<td>10</td>
<td>v1.20.5</td>
<td>v1.20</td>
<td>51</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4: Partial Update Tests – Partial updates applying a single commit that fixes a patch but each individual change from a version update is available.
line and multi-line patches for security vulnerabilities and other bugs in real programs. ATTUNE resolved references between modified and original executables and program state with binary transformations, but we cannot claim that ATTUNE’s set of transformations will resolve all types of references supported by the expansive x86-64 instruction set. We have not yet studied C++ or other non-C programs and we have not yet investigated ARM or other architectures. The bugs we studied may not be representative of real-world bugs; notably we have not yet studied GUI bugs.

Construct. The overhead measurements comparing ATTUNE to Klee are arguably unfair, since the symbolic execution explores “from scratch” even though, in principle, Klee’s symbolic execution engine could be modified to leverage rr’s verbose execution traces. We considered integrating Klee with record/replay to be a major research effort, outside the scope of this work. Zuo et al. [10] recently completed such an effort, going even further by skipping ATTUNE’s verbose re-recording entirely, and integrating Klee with lightweight hardware-assisted control and data tracing. Zuo et al. present what they call shepherded symbolic execution, where a new production release cycle is incurred whenever constraint solving bogs down while trying to match the lightly recorded trace. In each new production build, instrumentation is added to capture key data values involved in complex constraint dependencies (long chains of symbolic writes and accesses to large symbolic memory objects). Assuming the bug reoccurs sufficiently often in production, after several release cycles the shepherded symbolic execution will eventually find inputs that reproduce the bug (not necessarily the same inputs that triggered the bug when it was originally discovered). Of the thirteen bugs in Zuo et al.’s dataset, two were reproduced from the initial lightweight recording, while the other eleven required from 2 to 10 re-occurrences in production. Their paper did not specify the real-world calendar time involved, but we think it is safe to assume it was longer than the 60 minutes we allowed for Klee timeout.

38
4.5. Limitations

Our ATTUNE prototype extending rr inherits rr’s design decision to replay multi-threaded recordings on a single thread and simulate thread interleaving by interrupting that single thread’s execution [46, 47]. Although ATTUNE accommodates thread synchronization and faithfully emulates the error state, rr’s approach makes it impossible for ATTUNE to accurately verify patches for concurrency bugs that manifest due to the true parallelism of multi-core execution. There is nothing in ATTUNE itself that inherently prevents it from addressing concurrency bugs, but we would need to find a faithfully multi-threading replacement for rr, ATTUNE also relies on rr to re-record the execution trace in the user environment and to replay that recording in the developer environment with the original version of the program [46, 47, 18]. Since rr was designed to be used during developer testing, with too high overhead for production [46], we adopt the re-recording model shown in Figure 4. In theory, lightweight production recorders could fail to capture sufficient detail to faithfully replay some behaviors even in the same user environment, in which case the re-recording might not manifest the bug, but Mashtizadeh et al. [9] explain this limitation is generally unimportant in practice.

A few ATTUNE limitations are orthogonal to the rr recorder. ATTUNE does not currently verify patches to preprocessor macros, since it compares the source file versions rather than the results of preprocessing the source files. ATTUNE also does not currently support generating tests for patches that change the size of a data structure on the stack or in the heap. We allow new values to be put on the stack and heap, but don’t adjust memory allocation when replaying logged values.

Ideally, ATTUNE would address the privacy concerns inherent in all bug report systems that send information gathered in the user environment to the developer. This might be achieved by adding an anonymization phase during or after re-recording with rr, prior to sending to the developer. For example, we could use path conditions and a constraint solver to generate new anonymous data forcing the same execution paths, as was done in [48, 49]. Something
like trace wringing [50] might also be an option. We see anonymizing user environment recordings as a major engineering effort that is outside the scope of our research. However, we note that unlike third-party website session script recordings [51], ATTUNE does not run surreptitiously: the user has to select lightweight recordings to re-record and submit to the developer.

5. Related Work

iFixR [3] automatically generates candidate patches from bug reports, but relies on conventional regression testing even though those tests initially failed to detect the bug. In future work, we plan to investigate integrating ATTUNE with automatic program repair (APR) technology. Differential unit tests [52] construct unit tests using in-memory program state immediately prior to invoking the target method, but cannot reproduce bugs not detected by the original developer tests. [53] similarly extracts unit tests from developer execution traces. In future work, we will investigate constructing unit tests from the ad hoc tests generated by ATTUNE.

KATCH [54] combines symbolic execution with heuristics to generate test cases that cover the patched part of the code, while shadow symbol execution [55] symbolically explores divergences between original and patched versions. Neither leverages execution traces recorded in the user environment, nor fully models system calls, so the generated test cases may not reflect the bug-triggering circumstances. However, symbolic execution enables reaching parts of the program not exercised by the recording, complementing ATTUNE.

Parallel retro-logging [56] allows developers to change their logging instrumentation so previous executions produce augmented logs, but the program is not modified. Network-level traffic cloning tools can relay or replay the network inputs for service-oriented and microservices architectures. For example, in Parikshan [57] the traffic is fed to a forked copy of an architectural component in a sandbox, for debugging, or to a modified version of a component, for testing patches. But the replay is not necessarily faithful when there are other
sources of non-determinism besides network traffic

Kuchta et al. [55] generates tests for software patches using "shadow symbolic execution". The old program shadows the new version as the two are symbolically executed in tandem. Whenever new and old diverge, their Shadow tool generates a test exercising the divergence, to comprehensively test new behaviors. Shadow’s symbolic execution time budget might permit reaching parts of the program not exercised by available user execution, complementing ATTUNE. Shadow does not leverage user execution traces and may not model all system calls, so its tests may not reflect known bug-triggering user environments. It only considers control-flow divergences, not data-only divergences, whereas ATTUNE relies on the program and environment state (data) from the verbose re-recording. Further, as explained by Kuchta et al., Shadow suffers from the incompleteness of symbolic execution, the impact of the initial set of inputs, multi-hunk patches (several of our studied patches cross multiple files), and the technical limitations of building on top of KLEE and LLVM bitcode — external calls to native code, such as library and system calls, are challenging. ATTUNE assumes the faithful recording of these calls.

Elbaum et al. [52] introduced "differential unit tests" generated from the execution traces of developer system tests. Their CR (Carving and Replaying) tool extracts and combines the trace segments that construct in-memory program state as it was just prior to invoking the target Java method, which then serves as a unit test. CR also complements ATTUNE, since its system tests would likely exercise the program more broadly than available user execution traces. Since CR does not leverage user execution traces and its system traces support only in-memory events, its tests may not reflect known bug-triggering user environments. Other work similarly extracts unit tests from developer execution traces, e.g., [53], with analogous advantages and disadvantages. CR does not attempt to continue replay through the execution of the method under test, the method’s return to its caller in the full system execution, and beyond. In contrast, ATTUNE’s ad hoc tests are generated from system recordings made in user rather than developer environments, and the primary goal is indeed to
continue replay of the full system through the execution of every changed function until its clear the bug no longer manifests — a more challenging problem.

ATTUNE requires faithful replay as a baseline, including emulation of interactions with files, databases and other resources in the user environment, whereas CR replays only in-memory program state. Tiwari et al. [2] take a similar approach to Elbaum et al., but cull their unit tests from production executions rather than from developer system tests. Their focus is on devising test oracles for pseudo-tested methods, where the test suite exercises the methods but no test oracle specifies their expected behavior. As in our user environment validation, Tiwari et al. assume that previous executions in the production environment produced the desired results.

A problem posed by Kravets and Tsafrir [58] is more similar to ad hoc test generation. They proposed “mutable replay”, where a record-replay engine tries to execute a modified program by closely matching a recorded execution trace from a previous program version. They sketched a hypothetical design based on a then-recent record-replay system, Scribe [59].

The Kravets and Tsafrir paper motivated the Scribe developers to implement "mutable replay" themselves [60]. They leveraged checkpoint/restart [61] in a backtracking search algorithm that sought to minimize adds/deletes to the recorded event log. Although successful on many bug-fix examples in the sense that the “mutable replay” continued through the modified portion of the code, the constructed execution was not necessarily the same execution that would have occurred had the modified code been in place in the user environment at the time the original code encountered the bug, which is what ATTUNE aims. Scribe’s implementation centered on a special Linux kernel module that intercepted system calls and other non-deterministic events within the operating system kernel, granting complete control over how the kernel responded to the events, whereas ATTUNE runs without privileges in user-space with no changes to the operating system. Scribe required a shared file system (copy on write) between the recording and replaying environments, so was impractical for the post-deployment scenarios we envision, where no files, databases, etc. are shared.
between user and developer environment other than the contents accessed during
verbose re-recording and thus included in the log sent to developers.

There are numerous other record-replay tools in the literature, e.g., [62, 63, 64, 65]. Some versions of gdb build-in recording and replaying debugging ses-
sions [66], as does Microsoft’s IntelliTrace [67]. Some work trades off faithful
replay guarantees to better address long-lived latent bugs for time-travel debug-
ging [68, 69]. In some record-replay papers the recorded log is referred to as a
test case, but most replays only reproduce the buggy execution of the recorded
version of the program, and cannot be used to test patched versions.

Many record-replay tools focus on reproducing concurrency bugs, e.g., [70, 71, 72]. tsan11rec [73] combines a custom scheduler for detecting data races
with a sparse approach to record/replay: it records only those sources of non-
determinism configured per application. tsan11rec can record and replay I/O-
intensive software like video games, but cannot faithfully replay applications
where memory layout non-determinism significantly affects application behav-
ior. rr faithfully reproduces memory layout, but is not sufficiently perfor-
manent for I/O-bound applications – thus our re-recording architecture. While
ATTUNE supports ad hoc test generation for multi-threaded programs, our
prototype built on rr cannot generate tests for patches aimed specifically at
concurrency bugs due to how the rr implements multi-threading (it simulates
multiple threads within a single thread).

Much research focuses on reducing recording overhead, trading off lower
production overhead (thus better production performance) for faithful replay
guarantees, e.g., [74, 75, 76]. REPT [77] combines hardware tracing and binary
analysis to try to reconstruct execution traces, which can then be replayed with
the same program version. Castor [9] records multi-core applications by leverag-
ing hardware-optimized logging, transactional memory, and a custom compiler.
Its successful replays allow for slightly modified binaries that do not impact
program state. Zuo et al.’s approach outlined in Section 4.4, called Execution
Reconstruction (ER) [10], begins with hardware tracing of control and data flow.
After a failure, ER uses symbolic execution to find an input that is consistent
with the trace, i.e., reproduces the bug. When constraint solving bogs down, ER releases a production patch that records selected data values chosen to shepherd the symbolic execution further. This process iterates. If the failure occurs often enough, eventually sufficient production data will be accumulated to allow the symbolic execution to complete. Thus ER trades off lower production overhead for potentially quite long calendar-time delays in bug reproduction, and therefore bug repair. We assume some lightweight record/replay system for pervasive recording during production, even though none of them are guaranteed to reproduce every bug the first time it is encountered. ATTUNE’s re-recording with rr in the user environment kicks in only when the lightweight recorder succeeds in reproducing the bug, so could be quite prompt with REPT or Caster but would inherit ER’s wait time.

Although some papers about record-replay systems refer to capture-replay, e.g., [78], record-replay as discussed in this paper is different from most capture-playback tools. These record or script user actions to repeat for GUI compatibility testing across multiple operating system versions, browsers, or devices [79, 80, 81, 82, 83]. Capture-playback is conceptually similar to ad hoc test generation, but these tools focus on externally visible behavior and are not intended for faithful bug reproduction or testing patches.

Multi-Version Execution (MVE) provides an alternative approach to user validation. In MVE, the patched and original versions run simultaneously on production user workloads, adding runtime overhead but enabling immediate detection of undesirable divergences [84, 85, 86, 87]. In contrast, we envision that the user records production workloads with the old version and re-records offline as in Figure 2, but skips the developer stage and uses ATTUNE locally to generate ad hoc tests that replay the workloads with the patch. If all is satisfactory, production switches to the new version via some mechanism outside ATTUNE, e.g., live-update. Live-update tools deploy software updates without restarting running programs, e.g., by enabling the new version to resume a checkpoint from the old version similar to a fresh initialization [88, 89]. Dynamic software updating [90] combines multi-version execution with live-update, where
the update is applied to a forked copy while the original system continues to
operate. The new version shadows the original for a warmup period, and if
there are no problems production execution switches over. Unlike MVE systems
running different code versions, as in ATTUNE, LDX [91] runs two instances of
the same code to infer causality between events. The slave changes one event
from the master execution to find divergent impacts, orthogonal to our work.

Fuzzing seeks inputs that induce crashes and other problems [92, 93, 94, 95].
Other approaches also strive to induce bad behaviors, e.g., [96, 97]. Soltani
et al. [98] builds on EvoSuite’s search-based testing [99] to reproduce crashes.
Symbolic execution [54], fuzzing [100] and other approaches generate test suites
to achieve coverage goals. There is a rich literature concerned with generating
inputs intended to trigger or reproduce bugs. Generally, the same generated tests could be applied to multiple program versions — unless those tests
are "flaky". There has also been much work towards making tests repeatable,
which is sometimes difficult even in the developer environment on the exact
same system build [101]. These tools, as well as regression testing, complement
ATTUNE by providing generic testing methodologies, but ATTUNE’s targeted
approach based on the specific buggy execution provides a more efficient alternative. Compared to fuzzing, symbolic execution, and coverage based testing
ATTUNE targets the specific buggy execution without relying on approximate
heuristics like symbolic conditions and code coverage that cannot guarantee bug
reproduction.

Research in continuous integration and deployment techniques like those outlined by Shahin et al. in [102] provide a different functionality than ATTUNE.
Some deployment production environments even have test monitoring tools
built-in [103]. While they do incremental builds that test software during development, they do not provide the deploying users a chance to make changes
outside of what the developer has distributed.

Test case prioritization techniques like the one described by Srivastava et al. [104] and those described by Bajaj et al. [105] look to improve efficiency
in regression testing by looking to reduce the total number of tests when re-
gression bugs are introduced. While customer organizations could employ such
techniques to determine which commits they would like to introduce to their
distribution, we leave such discussions to future work. Other code analysis tech-
niques like software slicing [106, 107] and branch coverage [108, 109] that are
used to augment testing techniques provide a different function than ATTUNE.
Slicing identifies groups of statements that are most affected by a given change
to inform testing strategies. ATTUNE’s specialized tests remove the need for
software slicing as the log guarantees only the critical program paths are un-
der test. The popular branch coverage testing techniques are orthogonal to
ATTUNE as branch coverage is not a metric that is used by ATTUNE. Of
course ATTUNE’s test would cover specific branches of code, but maintaining
adequate coverage across the entire suite is still left to developer practices.

It should be noted that this technology is significantly different than auto-
mated program repair outlined by Le Goues et al. in [110] since we still rely on
the human developer to actually write the repair. It also differs from regression
testing techniques like those reviewed in [111] by Khatibsyarbibini et al.

Binary rewriting has been used for many reasons including implementing de-
defenses, automatic program repair, hot patching, and optimization. Hot patching
is an interesting example since it requires conserving dynamic program state at
the time the repair is applied, similar to binary quilting. Katana [112] has highly
sophisticated mechanisms for handling this problem, many of which would aug-
ment our current quilting procedure, but relies on trampolines to apply the
patches that could incur significant overhead the same as [113]. Other binary
rewriting mechanisms like Zipr [114, 115] raise the binary to a higher level IR
that allows for increased efficiency in the reassembly process similar to Egali-
to [17], but have demonstrated generic binary level defense transformations
instead of semantically complex bug specific patching.
6. Conclusion

ATTUNE (Ad hoc Test generation ThroUgh biNary rEwriting) leverages record-replay and binary rewriting technologies to automate test generation for security vulnerabilities and other critical bugs discovered post-deployment, when there are no existing tests for testing candidate patches, and little time for constructing and vetting new tests. ATTUNE first quilts the modified functions (the patch) into the original binary and then interprets the recorded execution trace from the original binary, as it executed in the user environment, to “replay” on the patched binary in the developer environment. The developer monitors the progress of the ad hoc test to check that the bug no longer manifests, but does not intervene in test generation and does not need to build test scaffolding.

We have augmented our original implementation of ATTUNE with binary patch decomposition, which integrates with the build process to give software operators (user organization IT staff) the ability to test and apply partial updates in the event the developer’s full release breaks functionality, e.g., because of incompatibilities with user environment infrastructure. Our BPD datastore lets operators selectively apply patches leaving most of the production version untouched. We showed that ATTUNE successfully generates tests for a wide range of known security vulnerabilities and other bugs in recent versions of open-source software, with minimal developer effort, both quickly and efficiently. We also demonstrated that BPD can successfully construct updated binaries to address installation-specific problematic behavior. Our open-source implementation is available at https://github.com/Programming-Systems-Lab/ATTUNE.

7. Acknowledgment

Funding: This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1563555 and CCF-1815494. We thank Yangruibo Ding, Victor Xu, and Ziao Wang for compiling the patch-testing dataset. We thank Robert O’Callahan and David Williams-King for their suggestions.
References

Representative of Field Behavior?, in: IEEE International Conference on
URL https://doi.org/10.1109/ICST.2017.36


Y. Le Traon, iFixR: Bug Report Driven Program Repair, in: 27th ACM
Joint Meeting on European Software Engineering Conference and Sympo-
posium on the Foundations of Software Engineering (ESEC/FSE), 2019,
pp. 314–325.
URL http://doi.acm.org/10.1145/3338906.3338935

[4] G. Catolino, F. Palomba, A. Zaidman, F. Ferrucci, Not All Bugs Are the
Same: Understanding, Characterizing, and Classifying the Root Cause of
S0164121219300536

cisa.gov/coordinated-vulnerability-disclosure-process (December 2019).

hackerone.com/disclosure-guidelines (July 2019).

M. Di Penta, D. Poshyvanyk, V. Ng, Assessing the Quality of the Steps
to Reproduce in Bug Reports, in: 27th ACM Joint Meeting on European


URL https://doi.org/10.1145/3428270

14 A. Ng, How the Equifax hack happened, and what still needs to be done (September 7 2018).
URL https://tinyurl.com/27633662

URL https://doi.org/10.1109/SCAM51674.2020.00018

URL https://doi.org/10.1145/3411502.3418424

URL https://doi.org/10.1145/3373376.3378470

URL https://rr-project.org/


URL http://klee.github.io/releases/


[34] ’cp -n -u’ and ’mv -n -u’ now consistently ignore the -u option, https://github.com/coreutils/coreutils/commit/7e244891b0c41bbf9f5b5917da71c183a8367ac (2018).


[74] A. Orso, B. Kennedy, Selective Capture and Replay of Program Executions, in: 3rd International Workshop on Dynamic Analysis (WODA),
2005, pp. 1—7.
URL https://doi.org/10.1145/1083246.1083251

URL http://doi.acm.org/10.1145/2814270.2814320

URL https://doi.org/10.1109/ICSM.2007.4362636

URL https://dl.acm.org/doi/10.5555/3291168.3291171

URL https://doi.org/10.1145/347324.348993

URL https://github.com/Microsoft/WinAppDriver

URL https://www.seleniumhq.org

URL http://appium.io/


URL https://doi.org/10.1109/ICSE.2019.00082

URL https://doi.org/10.1145/3338906.3338970

URL https://doi.org/10.1109/TSE.2018.2852744

URL https://doi.org/10.1109/ICSE.2017.27

URL https://doi.org/10.1145/2025113.2025179

URL https://doi.org/10.1145/3238147.3238176

URL https://doi.org/10.1145/3377811.3381749

URL https://doi.org/10.1109/ACCESS.2017.2685629


URL https://doi.org/10.1109/ACCESS.2019.2938260

URL https://doi.org/10.1109/ICST46399.2020.00014


Generates test cases from record-replay execution trace
Tests candidate fixes in addition to reproducing bug
Decomposes version update binaries into partial patches
Customer selectively tests and applies partial patches
Leverages binary rewriting without requiring source code
Anthony Saieva received his BA in Computer Science from Stonehill College, his BS in Computer Engineering from University of Notre Dame, and his MS in Computer Science from Columbia University. Anthony was a PhD student in Computer Science at Columbia University at the time this work was conducted.

Gail Kaiser is a Professor of Computer Science at Columbia University. She received her ScB in Computer Science and Engineering from Massachusetts Institute of Technology, her MS in Computer Science from Carnegie Mellon University, and her PhD in Computer Science from Carnegie Mellon University.
Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: