# Journal Pre-proof


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

Please cite this article as: A. Saieva and G. Kaiser, Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting. *The Journal of Systems & Software* (2022), doi: https://doi.org/10.1016/j.jss.2022.111381.

Title page with author details

# Update with Care: Testing Candidate Patches and Integrating Selective Updates

Anthony Saieva[a], Gail Kaiser[b]

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

[a] *ant@cs.columbia.edu*
[b] *kaiser@cs.columbia.edu (Corresponding author)*

 

# Update with Care: Testing Candidate Bug Fixes and Integrating Selective Updates through Binary Rewriting

Anthony Saieva[a], Gail Kaiser[b]

*Columbia University*
*Department of Computer Science*
*New York, NY 10027*

[a]*ant@cs.columbia.edu*
[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

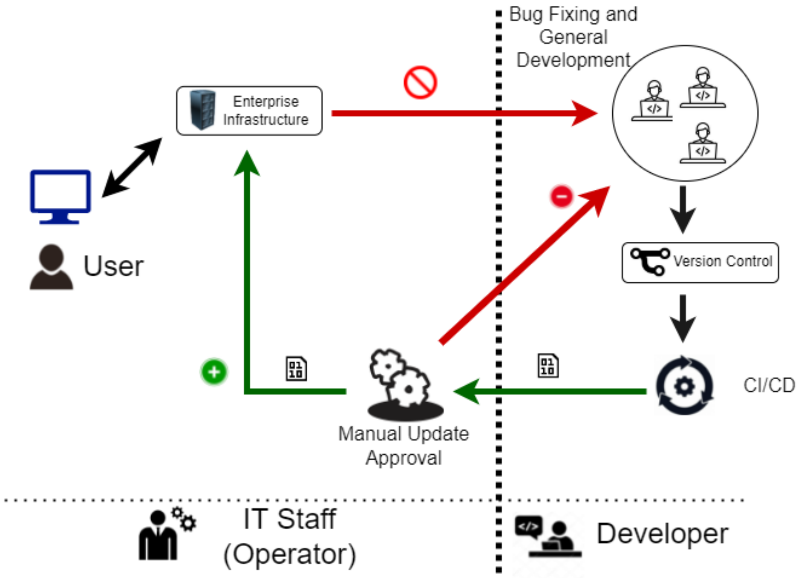


Figure 1: Ad hoc Test Generation Context

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.

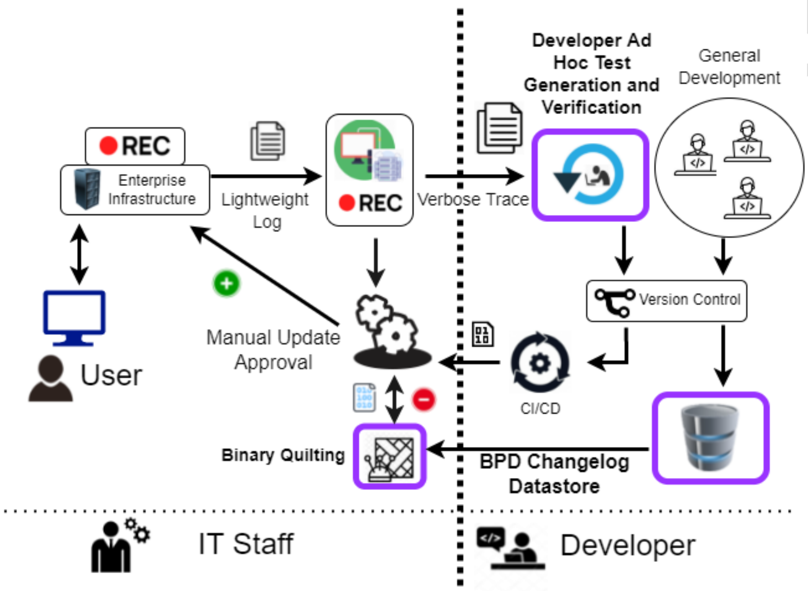


Figure 2: Ad hoc Test Generation Workflow

## 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 (**A**d hoc **T**est generation **T**hro**U**gh bi**N**ary r**E**writing), 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 reproduction. 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 datstore 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 process/thread layout. Nonetheless, bug reproduction is difficult. The premise of record-replay technology highlights the difficulty in capturing all of the conditions that led to erroneous behavior and recreating those conditions in the developer environment. Standard bug reports often fail to capture all the relevant 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 scaffolding 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 communication 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

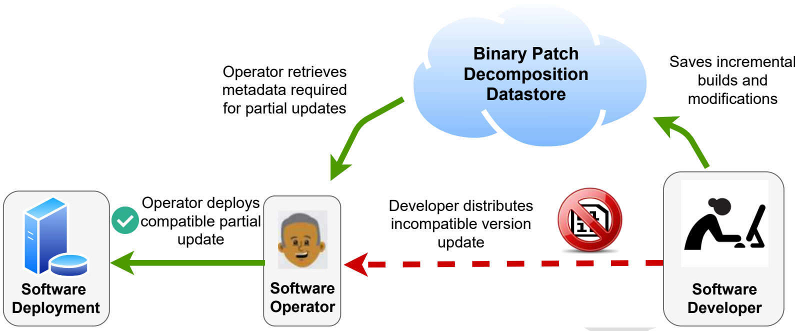


Figure 3: Ad hoc Test Generation and Partial Updates for Customer Organizations

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. Customer organizations tend to update software only when necessary for fear of updates introducing side effects that disrupt service. Software releases according 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 infrastructure 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

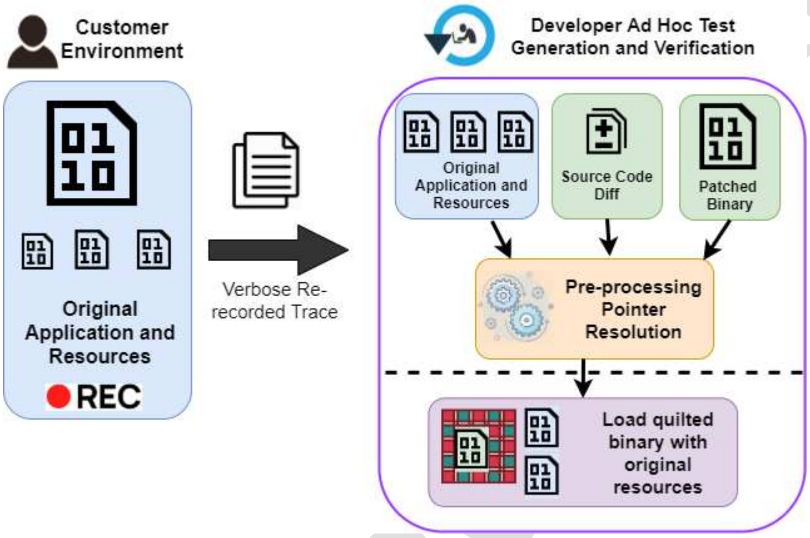


Figure 4: Recording and Preparation for Ad Hoc Test Generation

prototype implementation, portable across Linux distributions, is available at https://github.com/Programming-Systems-Lab/ATTUNE.

## 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 `mmap`ped 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 technique 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 allocation 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 example 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 binary/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

```
    ...
<c> 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] NS
0x0000403a  [3166, 0] NS
0x00004046  [3182, 0] NS
```

Figure 7: libpng-1 Relevant DWARF Line Entries

Two mechanisms enable this translation: The first is the symbol table standard in all ELF files and the 2nd is DWARF information. 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.

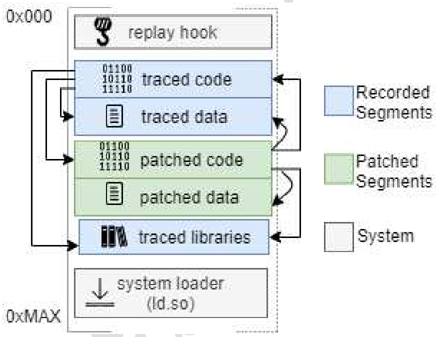


Figure 8: Address Space Detail

### *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
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
```

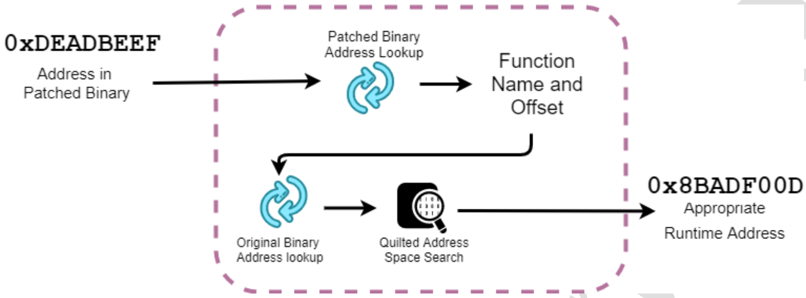


Figure 9: Pointer Translation Procedure

**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


Figure 11: PLT Transformation

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.

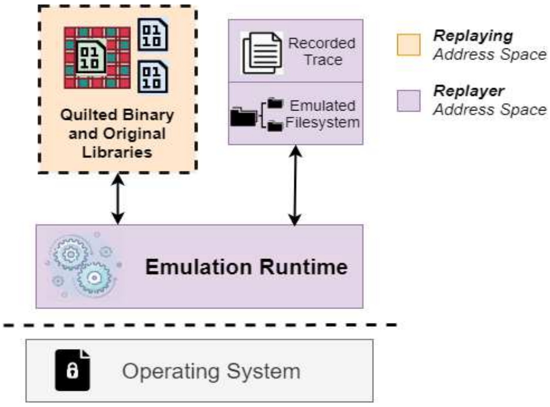


Figure 12: Runtime Architecture

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 delivery 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_operatione
     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

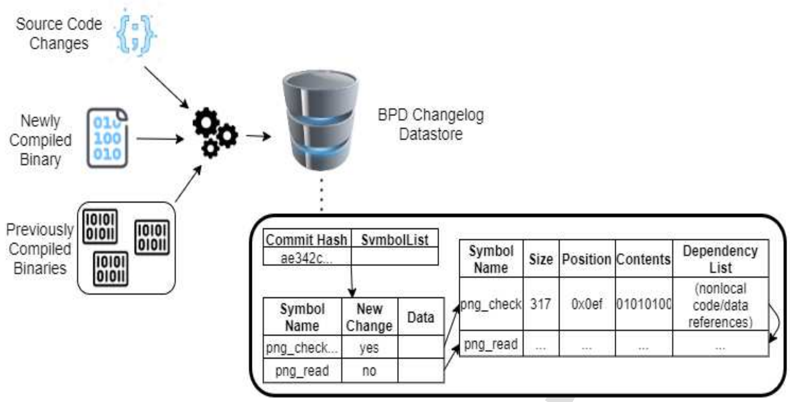


Figure 13: Binary Patch Decomposition Datastore

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 dependency 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 without leaving behind any software bloat or extra instructions that would impose performance problems. Effectively we are recompiling the binary to some intermediate build between version releases despite not having the source code. ATTUNE's binary rewriting technique avoids the need for recompilation making it more efficient by both saving compute cycles, and eliminating the need to store source code changes in the BPD datastore. Furthermore by not recompiling 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
    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 characters, e.g., the "#@" in *http://example.com#@evil.com* caused Curl to erroneously send a request to a malicious URL. The patch calls *sscanf* with a different filter string. Since the surrounding function handles all the URL parsing 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 errors 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]",
+     "%15[^\n:]:%3[/]%[^\n/?#]%[^\n]", /*new data*/
         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

```
+/* Return non zero if a non breaking space.  */
+ static int iswnbspace (wint_t wc) {
+  return ! posixly_correct && (wc == 0x00A0 ...
+ static int isnbspace (int c) {
+   return iswnbspace (btowc (c));
+}
+
wc (args) {
-  if (iswspace (wide_char))
+  if (iswspace (wide_char)
   || iswnbspace(wide_char))
        goto mb_word_separator;
        ...
-    if (isspace (to_uchar (p[-1])))
+    if (isspace (to_uchar (p[-1]))
+        || isnbspace (to_uchar (p[-1])))
+            goto word_separator;
}
...
```

Figure 17: wc-1 New Function and Refactoring

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 modified *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. Redis 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 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 challenging patches where the compiled binaries included complete metadata. However, 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 parameter 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 easily. 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 performance 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 execution, so it incurs some memory overhead at test time, as shown in Table 2. *need more here* Symbolic execution, on the other hand, requires significant resources to maintain the intermediate program states required to develop test cases. We found on the studied bugs that ATTUNE reduced memory overhead over 90% in all cases and could reduce memory usage by as much as 97% compared to KLEE.

### *4.3. Operators validate released patches with their own workloads and apply partial 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 broken. 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 environments. ATTUNE summarizes the "diffs" in source and binary code, and

| Bug | ATTUNE | KLEE | ATTUNE Time | KLEE Time | Speedup | ATTUNE Mem | KLEE Mem | Overhead Reduction |
|---|---|---|---|---|---|---|---|---|
| wc-1 [22] | ✓ | ✓ | 1.37s | 300.046s (5m) | **99.5%** | 5.9 KB | 108.388 KB | **94.5%** |
| wc-2 [32] | ✓ | ✗ | 1.277s | na | na | 2.8 KB | 107.7 KB | **97.4%** |
| yes-1 [25] | ✓ | ✓ | 3.4s | 8.569s | **60.3%** | 10.6 KB | 107.09 KB | **90.1%** |
| shred-1 [27] | ✗ | ✗ | na | na | na | na | na | na |
| ls-1 [33] | ✓ | ✓ | 1.6s | 19.57s | **91.82%** | 7.4 KB | 132.9 KB | **94.4%** |
| mv-1 [34] | ✓ | ✓ | 3.6s | 58.4s | **93.84%** | 4.3 KB | 208.2 KB | **97%** |
| df-1 [26] | ✓ | ✓ | 1.48s | 18.869s | **92.15%** | 5.97 KB | 151 KB | **96.05%** |
| bs-1 [35] | ✓ | ✗ | 1.2s | na | na | 5.6 KB | 113.37 KB | **95.06%** |

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 *httperf* benchmarking tool [37] making thousands of connections. Similar to the redis 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 decision 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. Arguably, 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 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 benchmark, 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-

| Bug | Data Resolutions | Code Resolutions | Buggy Version Tag | Patch Version Tag | Distinct Changes Between Versions | Partial Update Success |
|---|---|---|---|---|---|---|
| curl-1 [20] | 4 | 31 | curl_7_63_0 | curl_7_64_0 | 128 | ✓ |
| curl-2 [38] | 69 | 318 | curl_7_63_0 | curl_7_64_0 | 128 | ✓ |
| curl-5 [39] | 6 | 53 | curl_7_33_0 | curl_7_34_0 | 246 | ✓ |
| curl-6 [40] | 6 | 71 | curl_7_63_0 | curl_7_64_0 | 128 | ✓ |
| curl-8 [28] | n/a | n/a | curl_7_61_0 | curl_7_60_0 | 223 | ✗ |
| curl-9 [41] | 8 | 26 | curl_7_62_0 | curl_7_63_0 | 122 | ✓ |
| curl-10 [42] | 3 | 21 | curl_7_62_0 | curl_7_63_0 | 122 | ✓ |
| curl-11 [43] | 273 | 1012 | curl_7_62_0 | curl_7_63_0 | 122 | ✓ |
| curl-12 [21] | 37 | 103 | curl_7_50_0 | curl_7_51_0 | 333 | ✓ |
| libpng-1 [19] | 1 | 6 | v1.6.34 | v1.6.35 | 53 | ✓ |
| libpng-2 [44] | 1 | 6 | v1.6.32beta02 | v1.6.33beta02 | 97 | ✓ |
| wc-1 [22] | 109 | 298 | v8.30 | v8.31 | 90 | ✓ |
| wc-2 [32] | 79 | 155 | v8.26 | v8.27 | 69 | ✓ |
| yes-1 [25] | 234 | 399 | v8.30 | v8.31 | 90 | ✓ |
| shred-1 [27] | n/a | n/a | v8.27 | v8.28 | 72 | ✗ |
| ls-1 [33] | 380 | 387 | v8.29 | v8.30 | 68 | ✓ |
| mv-1 [34] | 89 | 204 | v8.29 | v8.30 | 68 | ✓ |
| df-1 [26] | 164 | 348 | v8.28 | v8.29 | 65 | ✓ |
| bs-1 [35] | 140 | 296 | v8.28 | v8.29 | 65 | ✓ |
| wget-1 [45] | 8 | 16 | 5.0.6 | 5.0.7 | 30 | ✓ |
| redis-1 [23] | 3 | 10 | v1.19.5 | v1.20 | 51 | ✓ |

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.

### 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 sessions [66], as does Microsoft's IntelliTrace [67]. Some work trades off faithful replay guarantees to better address long-lived latent bugs for time-travel debugging [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 behavior. rr faithfully reproduces memory layout, but is not sufficiently performant 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 leveraging 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 regression 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 techniques 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 effected 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 under 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 automated 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 Khatibsyarbini et al.

Binary rewriting has been used for many reasons including implementing 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 augment 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 Egalito [17], but have demonstrated generic binary level defense transformations instead of semantically complex bug specific patching.

## 6. Conclusion

ATTUNE (**A**d hoc **T**est generation **T**hro**U**gh bi**N**ary r**E**writing) 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

[1] Q. Wang, Y. Brun, A. Orso, Behavioral Execution Comparison: Are Tests Representative of Field Behavior?, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 321–332. URL https://doi.org/10.1109/ICST.2017.36

[2] D. Tiwari, L. Zhang, M. Monperrus, B. Baudry, Production Monitoring to Improve Test Suites (December 2020). arXiv:2012.01198. URL https://arxiv.org/abs/2012.01198

[3] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, Y. Le Traon, iFixR: Bug Report Driven Program Repair, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium 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 Bugs, Journal of Systems and Software 152 (2019) 165–181. URL http://www.sciencedirect.com/science/article/pii/S0164121219300536

[5] Cybersecrity & Infrastructure Security Agency, CISA COORDINATED VULNERABILITY DISCLOSURE (CVD) PROCESS, https://www.cisa.gov/coordinated-vulnerability-disclosure-process (December 2019).

[6] hackerone, Vulnerability Disclosure Guidelines, https://www.hackerone.com/disclosure-guidelines (July 2019).

[7] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, 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

Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 86–96.
URL http://doi.acm.org/10.1145/3338906.3338947

[8] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, G. Wang, Understanding the Reproducibility of Crowd-reported Security Vulnerabilities, in: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore, MD, 2018, pp. 919–936.
URL https://www.usenix.org/conference/usenixsecurity18/presentation/mu

[9] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, M. Rosenblum, Towards Practical Default-On Multi-Core Record/Replay, in: 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017, pp. 693–708.
URL http://doi.acm.org/10.1145/3037697.3037751

[10] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, B. Kasikci, Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1155–1170. doi:10.1145/3453483.3454101.
URL https://doi.org/10.1145/3453483.3454101

[11] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, S. Ducasse, Untangling Fine-Grained Code Changes, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 341–350.
URL https://doi.org/10.1109/SANER.2015.7081844

[12] J. Tucek, W. Xiong, Y. Zhou, Efficient Online Validation with Delta Execution, in: 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009, pp.

193–204.
URL http://doi.acm.org/10.1145/1508244.1508267

[13] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, J. Bell, A Large-Scale Longitudinal Study of Flaky Tests, Proceedings of the ACM on Programming Languages 4 (OOPSLA) (2020) 1–29.
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

[15] A. Saieva, S. Singh, G. Kaiser, Ad hoc Test Generation Through Binary Rewriting, in: IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 115–126.
URL https://doi.org/10.1109/SCAM51674.2020.00018

[16] A. Saieva, G. Kaiser, Binary Quilting to Generate Patched Executables without Compilation, in: Workshop on Forming an Ecosystem Around Software Transformation (FEAST), 2020, pp. 3—-8.
URL https://doi.org/10.1145/3411502.3418424

[17] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, V. P. Kemerlis, Egalito: Layout-Agnostic Binary Recompilation, in: 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, pp. 133—-147.
URL https://doi.org/10.1145/3373376.3378470

[18] Mozilla, what rr does (2021).
URL https://rr-project.org/

[19] Red Hat Bugzilla – Bug 1599943, libpng: Integer overflow and resultant divide-by-zero, https://bugzilla.redhat.com/show_bug.cgi?id=

1599943, CVE: https://nvd.nist.gov/vuln/detail/CVE-2018-13785 (2019).

[20] Curl string parsing bug, https://github.com/curl/curl/pull/3365 (2019).

[21] Curl string parsing vulnerability, https://github.com/curl/curl/commit/3bb273db7, CVE: https://curl.haxx.se/docs/CVE-2016-8624.html (2019).

[22] wc special character bug, https://github.com/coreutils/coreutils/commit/a5202bd58531923e (2019).

[23] Redis monitor request causes crash, https://github.com/antirez/redis/commit/e2c1f80b (2019).

[24] wget insert new loop to parse URL's, https://github.com/mirror/wget/commit/4d729e322fae, CVE: https://nvd.nist.gov/vuln/detail/CVE-2017-6508. (2019).

[25] yes coreutils library function, https://github.com/coreutils/coreutils/commit/44af84263e (2019).

[26] df coreutils library function, https://github.com/coreutils/coreutils/commit/b04ce61958c (2019).

[27] shred coreutils library function, https://github.com/coreutils/coreutils/commit/c34f8d5c787e6 (2019).

[28] Curl change parameter fix, https://github.com/curl/curl/commit/e50a2002 (2019).

[29] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, in: 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), 2008, pp. 209—-224.

URL https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar_html/index.html

[30] KLEE Team, KLEE LLVM execution engine, http://klee.github.io/ (2021).

[31] KLEE Team, Stable releases of KLEE (2021). URL http://klee.github.io/releases/

[32] wc reports wrong byte counts when using '–from-files0=-', https://debbugs.gnu.org/cgi/bugreport.cgi?bug=23073 (2016).

[33] ls -aA shows . and .. in an empty directory, https://debbugs.gnu.org/cgi/bugreport.cgi?bug=30963 (2018).

[34] 'cp -n -u' and 'mv -n -u' now consistently ignore the -u option, https://github.com/coreutils/coreutils/commit/7e244891b0c41bbf9f5b5917d1a71c183a8367ac (2018).

[35] Running b2sum with –check option, and simply providing a string "BLAKE2", https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28860 (2017).

[36] Redis benchmark tests server functionality, https://github.com/antirez/redis (2019).

[37] The httperf HTTP load generator, https://github.com/httperf (2019).

[38] Curl string parsing bug, https://github.com/curl/curl/pull/3381 (2019).

[39] Curl info leak, https://github.com/curl/curl/pull/3381, cVE: https://curl.haxx.se/docs/CVE-2017-1000101.html (2019).

[40] Curl security vulnerability, https://github.com/curl/curl/pull/3433 (2019).

[41] Curl_follow: accept non-supported schemes for ”fake” redirects, https://github.com/curl/curl/commit/2c5ec339ea67f43ac370ae77636a0f915cc5fbeb (2018).

[42] URL: fix IPv6 numeral address parser, https://github.com/curl/curl/pull/3219 (2018).

[43] Curl globbing error, https://github.com/curl/curl/issues/3251 (2019).

[44] libpng IDAT miscalculation, https://sourceforge.net/p/libpng/bugs/270/ (2019).

[45] Simple fix stops creating the log when using -o and -q in the background, https://github.com/mirror/wget/commit/7ddcebd61e170fb03d361f82bf8f5550ee62a1ae (2018).

[46] R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, N. Partush, Engineering Record and Replay for Deployability, in: USENIX Annual Technical Conference (USENIX ATC), 2017, pp. 377–389.
URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[47] R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, N. Partush, Engineering Record And Replay For Deployability: Extended Technical Report (May 2017).
URL http://arxiv.org/abs/1705.05937

[48] M. Castro, M. Costa, J.-P. Martin, Better Bug Reporting with Better Privacy, in: 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008, pp. 319—-328.
URL https://doi.org/10.1145/1346281.1346322

[49] J. Clause, A. Orso, Camouflage: Automated Anonymization of Field Data, in: 33rd International Conference on Software Engineering (ICSE), 2011,

pp. 21—-30.
URL https://doi.org/10.1145/1985793.1985797

[50] D. Dangwal, W. Cui, J. McMahan, T. Sherwood, Safer Program Behavior Sharing Through Trace Wringing, in: 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 1059—-1072.
URL https://doi.org/10.1145/3297858.3304074

[51] S. Englehardt, G. Acar, A. Narayanan, No boundaries: Exfiltration of personal data by session-replay scripts, Freedom to Tinker.
URL https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/

[52] S. Elbaum, H. N. Chin, M. B. Dwyer, M. Jorde, Carving and Replaying Differential Unit Test Cases from System Test Cases, IEEE Transactions on Software Engineering (TSE) 35 (1) (2009) 29–45.
URL https://doi.org/10.1109/TSE.2008.103

[53] F. Křikava, J. Vitek, Tests from Traces: Automated Unit Test Extraction for R, in: 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2018, pp. 232—-241.
URL https://doi.org/10.1145/3213846.3213863

[54] P. D. Marinescu, C. Cadar, KATCH: High-Coverage Testing of Software Patches, in: 9th Joint Meeting of the European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2013, pp. 235–245.
URL https://doi.org/10.1145/2491411.2491438

[55] T. Kuchta, H. Palikareva, C. Cadar, Shadow Symbolic Execution for testing software patches, ACM Transactions on Software Engineering and Methodology (TOSEM) 27 (3) (2018) 10:1–10:32.
URL http://doi.acm.org/10.1145/3208952

[56] A. Quinn, J. Flinn, M. Cafarella, Sledgehammer: Cluster-fueled debugging, in: 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), 2018, pp. 545–560.
URL https://dl.acm.org/doi/10.5555/3291168.3291208

[57] N. Arora, J. Bell, F. Ivančić, G. Kaiser, B. Ray, Replay Without Recording of Production Bugs for Service Oriented Applications, in: 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), 2018, pp. 452–463.
URL http://doi.acm.org/10.1145/3238147.3238186

[58] I. Kravets, D. Tsafrir, Feasibility of Mutable Replay for Automated Regression Testing of Security Updates, in: 2nd Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE), 2012, pp. 1–6.
URL http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session4_paper2.pdf

[59] O. Laadan, N. Viennot, J. Nieh, Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems, in: ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2010, pp. 155–166.
URL http://doi.acm.org/10.1145/1811039.1811057

[60] N. Viennot, S. Nair, J. Nieh, Transparent Mutable Replay for Multicore Debugging and Patch Validation, in: 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013, pp. 127–138.
URL http://doi.acm.org/10.1145/2451116.2451130

[61] O. Laadan, J. Nieh, Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems, in: USENIX Annual Technical Conference (ATC), 2007, pp. 25:1–25:14.
URL https://dl.acm.org/doi/10.5555/1364385.1364410

[62] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, W. G. J. Halfond, ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports, in: 41st International Conference on Software Engineering (ICSE), 2019, pp. 128–139.
URL `https://doi.org/10.1109/ICSE.2019.00030`

[63] E. Pobee, W. K. Chan, AggrePlay: Efficient Record and Replay of Multithreaded Programs, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 567–577.
URL `http://doi.acm.org/10.1145/3338906.3338959`

[64] H. Liu, S. Silvestro, W. Wang, C. Tian, T. Liu, iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications, in: 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2018, pp. 344–358.
URL `http://doi.acm.org/10.1145/3192366.3192380`

[65] Y. Shalabi, M. Yan, N. Honarmand, R. B. Lee, J. Torrellas, Record-Replay Architecture as a General Security Framework, in: IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 180–193.
URL `https://doi.org/10.1109/HPCA.2018.00025`

[66] GDB Wiki, Process Record and Replay (2013).
URL `https://sourceware.org/gdb/wiki/ProcessRecord`

[67] Microsoft, IntelliTrace for Visual Studio Enterprise (C#, Visual Basic, C++) (2018).
URL `https://docs.microsoft.com/en-us/visualstudio/debugger/intellitrace?view=vs-2019`

[68] A. Miraglia, D. Vogt, H. Bos, A. Tanenbaum, C. Giuffrida, Peeking into the Past: Efficient Checkpoint-Assisted Time-Traveling Debugging, in:

2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 455–466.
URL https://doi.org/10.1109/ISSRE.2016.9

[69] D. Vogt, Efficient High Frequency Checkpointing for Recovery and Debugging, Ph.D. thesis, Vrije Universiteit Amsterdam (2019).
URL https://research.vu.nl/ws/portalfiles/portal/77028965/complete+dissertation.pdf

[70] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, B. Liang, Detecting Concurrency Memory Corruption Vulnerabilities, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 706—-717.
URL https://doi.org/10.1145/3338906.3338927

[71] Y. Hu, I. Neamtiu, A. Alavi, Automatically Verifying and Reproducing Event-Based Races in Android Apps, in: 25th International Symposium on Software Testing and Analysis (ISSTA), 2016, pp. 377–388.
URL http://doi.acm.org/10.1145/2931037.2931069

[72] S. Rattanasuksun, T. Yu, W. Srisa-An, G. Rothermel, RRF: A Race Reproduction Framework for Use in Debugging Process-Level Races, in: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 162–172.
URL https://doi.org/10.1109/ISSRE.2016.35

[73] C. Lidbury, A. F. Donaldson, Sparse Record and Replay with Controlled Scheduling, in: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2019, pp. 576–593.
URL http://doi.acm.org/10.1145/3314221.3314635

[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

[75] Y. Hu, T. Azim, I. Neamtiu, Versatile yet Lightweight Record-and-Replay for Android, in: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015, pp. 349–366.
URL http://doi.acm.org/10.1145/2814270.2814320

[76] S. Joshi, A. Orso, SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions, in: 23rd IEEE International Conference on Software Maintenance (ICSM), 2007, pp. 234–243.
URL https://doi.org/10.1109/ICSM.2007.4362636

[77] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, I. Yun, REPT: Reverse Debugging of Failures in Deployed Software, in: 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), 2018, pp. 17–32.
URL https://dl.acm.org/doi/10.5555/3291168.3291171

[78] J. Steven, P. Chandra, B. Fleck, A. Podgurski, JRapture: A Capture/Replay Tool for Observation-Based Testing, in: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2000, pp. 158—-167.
URL https://doi.org/10.1145/347324.348993

[79] Microsoft, WinAppDriver (2021).
URL https://github.com/Microsoft/WinAppDriver

[80] SeleniumHQ, Browser automation (2021).
URL https://www.seleniumhq.org

[81] appium, Automation for apps (2021).
URL http://appium.io/

[82] S. Negara, N. Esfahani, R. P. L. Buse, Practical Android Test Recording with Espresso Test Recorder, in: 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 193–202.
URL https://doi.org/10.1109/ICSE-SEIP.2019.00029

[83] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, L. Ziarek, Mimic: Ui compatibility testing system for Android apps, in: 41st International Conference on Software Engineering (ICSE), 2019, pp. 246–256.
URL https://doi.org/10.1109/ICSE.2019.00040

[84] P. Hosek, C. Cadar, Safe Software Updates via Multi-version Execution, in: International Conference on Software Engineering (ICSE), 2013, pp. 612–621.
URL https://dl.acm.org/doi/10.5555/2486788.2486869

[85] P. Hosek, C. Cadar, VARAN the Unbelievable: An Efficient N-version Execution Framework, in: 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015, pp. 339–353.
URL http://doi.acm.org/10.1145/2694344.2694390

[86] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, D. Xu, Dual Execution for On the Fly Fine Grained Execution Comparison, in: 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015, pp. 325–338.
URL http://doi.acm.org/10.1145/2694344.2694394

[87] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, C. Giuffrida, kMVX: Detecting Kernel Information Leaks with Multi-variant Execution, in: 24th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), 2019, pp. 559–572.
URL http://doi.acm.org/10.1145/3297858.3304054

[88] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, A. S. Tanenbaum, Automating Live Update for Generic Server Programs, IEEE Transactions on Software Engineering 43 (3) (2017) 207–225.
URL https://doi.org/10.1109/TSE.2016.2584066

[89] S. Kashyap, C. Min, B. Lee, T. Kim, P. Emelyanov, Instant OS Updates via Userspace Checkpoint-and-Restart, in: USENIX Annual Technical Conference (USENIX ATC), 2016, pp. 605–619.
URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kashyap

[90] L. Pina, A. Andronidis, M. Hicks, C. Cadar, MVEDSUA: Higher availability dynamic software updates via multi-version execution, in: 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 573–585.
URL http://doi.acm.org/10.1145/3297858.3304063

[91] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, D. Xu, LDX: Causality Inference by Lightweight Dual Execution, in: 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016, pp. 503—-515.
URL https://doi.org/10.1145/2872362.2872395

[92] R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar, FuzzFactory: Domain-Specific Fuzzing with Waypoints, Proceedings of the ACM on Programming Languages (PACMPL) (2019) 3.
URL https://doi.org/10.1145/3360600

[93] L. Lampropoulos, M. Hicks, B. C. Pierce, Coverage Guided, Property Based Testing, Proceedings of the ACM on Programming Languages (PACMPL) (2019) 3.
URL https://doi.org/10.1145/3360607

[94] C. Lemieux, K. Sen, FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage, in: 33rd ACM/IEEE International

Conference on Automated Software Engineering (ASE), 2018, pp. 475–485.
URL http://doi.acm.org/10.1145/3238147.3238176

[95] J. Choi, J. Jang, C. Han, S. K. Cha, Grey-box Concolic Testing on Binary Code, in: 41st International Conference on Software Engineering (ICSE), 2019, pp. 736–747.
URL https://doi.org/10.1109/ICSE.2019.00082

[96] M. Biagiola, A. Stocco, F. Ricca, P. Tonella, Diversity-based Web Test Generation, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 142–153.
URL http://doi.acm.org/10.1145/3338906.3338970

[97] H. Wu, N. Changhai, J. Petke, Y. Jia, M. Harman, An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing, IEEE Transactions on Software Engineering (TSE) 46 (3) (2018) 302–320.
URL https://doi.org/10.1109/TSE.2018.2852744

[98] M. Soltani, A. Panichella, A. van Deursen, A Guided Genetic Algorithm for Automated Crash Reproduction, in: 39th International Conference on Software Engineering (ICSE), 2017, pp. 209—-220.
URL https://doi.org/10.1109/ICSE.2017.27

[99] G. Fraser, A. Arcuri, EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, in: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE), 2011, pp. 416–419.
URL https://doi.org/10.1145/2025113.2025179

[100] C. Lemieux, K. Sen, FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage, in: Proceedings of the 33rd

ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, 2018, p. 475–485.
URL https://doi.org/10.1145/3238147.3238176

[101] W. Lam, K. Muşlu, H. Sajnani, S. Thummalapenta, A Study on the Lifecycle of Flaky Tests, in: 42nd International Conference on Software Engineering (ICSE), 2020, pp. 1471–1482.
URL https://doi.org/10.1145/3377811.3381749

[102] M. Shahin, M. Ali Babar, L. Zhu, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, IEEE Access 5 (2017) 3909–3943.
URL https://doi.org/10.1109/ACCESS.2017.2685629

[103] D. Tiwari, L. Zhang, M. Monperrus, B. Baudry, Production Monitoring to Improve Test Suites, IEEE Transactions on Reliability (2021) 1–17.
URL https://ieeexplore.ieee.org/document/9526340

[104] P. R. Srivastava, TEST CASE PRIORITIZATION, Journal of Theoretical & Applied Information Technology 4 (3).
URL https://www.researchgate.net/profile/Dr-Praveen-Srivastava/publication/235799411_Test_case_prioritization/links/0c960531bf241d5bee000000/Test-case-prioritization.pdf

[105] A. Bajaj, O. P. Sangwan, A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms, IEEE Access 7 (2019) 126355–126375.
URL https://doi.org/10.1109/ACCESS.2019.2938260

[106] X. Li, A. Orso, More Accurate Dynamic Slicing for Better Supporting Software Debugging, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 28–38.
URL https://doi.org/10.1109/ICST46399.2020.00014

[107] B. Stoica, S. K. Sahoo, J. R. Larus, V. S. Adve, Statistical Program Slicing: a Hybrid Slicing Technique for Analyzing Deployed Software, CoRR abs/2201.00060. arXiv:2201.00060.
URL https://arxiv.org/abs/2201.00060

[108] S. Chatterjee, A. Shukla, A unified approach of testing coverage-based software reliability growth modelling with fault detection probability, imperfect debugging, and change point, Journal of Software: Evolution and Process 31 (3) (2019) e2150.
URL https://doi.org/10.1002/smr.2150

[109] Q. Yang, J. J. Li, D. M. Weiss, A Survey of Coverage Based Testing Tools, The Computer Journal 52 (5) (2009) 589–597.
URL https://doi.org/10.1093/comjnl/bxm021

[110] C. Le Goues, M. Pradel, A. Roychoudhury, Automated Program Repair, Communications of the ACM 62 (12) (2019) 56–65.
URL https://doi.org/10.1145/3318162

[111] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74–93.
URL https://doi.org/10.1016/j.infsof.2017.08.014

[112] A. Ramaswamy, S. Bratus, S. W. Smith, M. E. Locasto, Katana: A Hot Patching Framework for ELF Executables, in: International Conference on Availability, Reliability and Security (ARES), 2010, pp. 507–512.
URL https://doi.org/10.1109/ARES.2010.112

[113] H. Jeong, J. Baik, K. Kang, Functional Level Hot-patching Platform for. Executable and Linkable Format Binaries, in: IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017, pp. 489–494.
URL https://doi.org/10.1109/SMC.2017.8122653

[114] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, J. W. Davidson, Zipr: Efficient Static Binary Rewriting for Security, in: 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2017, pp. 559–566.
URL https://doi.org/10.1109/DSN.2017.27

[115] J. Hiser, A. Nguyen-Tuong, W. Hawkins, M. McGill, M. Co, J. Davidson, Zipr++: Exceptional Binary Rewriting, in: Workshop on Forming an Ecosystem Around Software Transformation (FEAST), 2017, pp. 9–15.
URL https://doi.org/10.1145/3141235.3141240

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

Biography

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: