Ad hoc Test Generation Through Binary Rewriting

Anonymous Author(s)

ABSTRACT

When a security vulnerability or other critical bug is not detected by the developers’ test suite, and is discovered post-deployment, developers must quickly devise a new test that reproduces the buggy behavior. Then the developers need to test whether their candidate patch indeed fixes the bug, without breaking other functionality, while racing to deploy before cyberattackers pounce on exposed user installations. This can be challenging when the bug discovery was due to factors that arose, perhaps transiently, in a specific user environment. If recording execution traces when the bad behavior occurred, record-replay technology faithfully replays the execution, in the developer environment, as if the program were executing in that user environment under the same conditions as the bug manifested. This includes intermediate program states dependent on system calls, memory layout, etc. as well as any externally-visible behavior. So the bug is reproduced, and many modern record-replay tools also integrate bug reproduction with interactive debuggers to help locate the root cause, but how do developers check whether their patch indeed eliminates the bug under those same conditions?

State-of-the-art record-replay does not support replaying candidate patches that modify the program in ways that diverge program state from the original recording, but successful repairs necessarily diverge so the bug no longer manifests. This work builds on record-replay, and binary rewriting, to automatically generate and run tests for candidate patches. These tests reflect the arbitrary (ad hoc) user and system circumstances that uncovered the vulnerability, to check whether a patch indeed closes the vulnerability but does not modify the corresponding segment of the program’s core semantics. Unlike conventional ad hoc testing, each test is reproducible and can be applied to as many prospective patches as needed until developers are satisfied. The proposed approach also enables users to make new recordings of her own workloads with the original version of the program, and automatically generate and run the corresponding ad hoc tests on the patched version, to validate that the patch does not introduce new problems before adopting.

CCS CONCEPTS

• Software and its engineering → Dynamic analysis; Automated static analysis; Software testing and debugging; Software post-development issues; Software reverse engineering;

KEYWORDS

test generation, software patching, record-replay, binary rewriting, security vulnerabilities

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
© 2020 Copyright held by the owner/author(s).

ACM ISBN yyyy
https://doi.org/xxxx

1 INTRODUCTION

When a security vulnerability or other critical bug is not detected by the developer test suite prior to deployment [64], but reported after deployment, it can be difficult and time-consuming for developers to construct new tests that reproduce the bug. Furthermore, the new tests need to verify that candidate patches do not exhibit the same or similar buggy behavior. Although minimizing the time from bug discovery to patch release is of the essence, users are wary of rushed patches, since they may break mission-critical functionality [67]. However, user validation is also difficult and time-consuming.

The existence of the Common Vulnerabilities and Exposures (CVE) list for security vulnerabilities [60] and mundane user bug reports [16, 20] demonstrate that not all bugs are discovered by developer tests. While vulnerability disclosures and bug reports sometimes include an explicit test sufficient both to reproduce the bug and to verify patches [30, 37], it is common for no such test to be known [21]. But disclosures and other bug reports often do include some evidence of the bug, such as memory dumps, stack traces, system logs, error messages, screenshots, and so on.

This paper presents a novel approach for rapidly generating new tests that reproduce the bug, support debugging, and verify that candidate patches do not exhibit buggy behavior, when the bug report includes a detailed execution trace demonstrating the bug. The approach even aids user validation of released patches.

The problem is illustrated abstractly in Figure 1. The top shows the original execution trace where the bug manifests. The bottom shows an automatically generated test emulating that execution trace. If the test is applied to the same code that produced the original execution, the emulated execution will be the same. If the test is applied to modified code, i.e., patched to try to fix the bug, it should emulate execution as if the modified code had been running instead of the original code. If the patch successfully fixes the bug, then this emulated execution will not manifest the bug. We refer to this concept as ad hoc test generation because the generated test emulates whatever user activities manifested the bug. Ad hoc test generation is feasible because security and other patches solely to fix bugs tend to be modest in size and scope, rarely changing core program semantics, shared memory layout or process/thread layout [61], so the execution trace divergence is not large. The significance lies in the shift from carefully planned and designed developer tests in time-crunch scenarios.

Current record-replay technology replays the original execution trace with the original code, but cannot replay that execution trace with patched code that affects program state. In particular,
TTUNE

An overview of ATTUNE’s workflow is shown in Figure 2. Since ATTUNE requires detailed traces that would impose too much time and space overhead for always-on recording in user environments, we envision that always-on recording is performed by a lightweight record-replay mechanism like Castor [46]. Then when user observation, analysis, monitoring, etc. determines that some portion of a lightweight trace manifests a security vulnerability or other bug, then (offline from production) that portion is segmented, re-played in the user environment, and re-recorded (similar to [22]), by ATTUNE’s verbose recorder (rr’s recorder in the prototype). To address the privacy concerns inherent in all bug-report systems that send information gathered in the user environment to the developer, sensitive data could be anonymized during this offline process (see [19, 23, 25]), and only the detailed but anonymous trace sent to developers. Unlike third-party website-session script recordings [27], the user must turn on recording (or opt into default-on) and submit traces; neither ATTUNE nor rr runs surreptitiously. Henceforth in this paper we elide these steps and simply refer to the re-recorded (verbose) trace as the original recording.

We explain the technical details of our approach and its limitations in Section 2. Our evaluation in Section 3 describes how a developer could use ATTUNE to test candidate patches for a variety of CVE security vulnerabilities and bugs from well-known open-source projects. Section 3 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. The section ends by analyzing threats to validity. Section 4 compares ad hoc test generation to related work. The paper concludes with a summary. The paper’s final version will link to our github repository, which includes download and record/replay scripts for our dataset as well as ATTUNE’s open-source implementation.

The contributions of this paper are:

- An architecture and design leveraging record-replay technology and binary rewriting to generate ad hoc test cases that aim to exercise candidate patches as if they had been executing in the user environment, instead of the previous buggy version, when the bad behavior was originally recorded.
- A technique for adding developer environment metadata to patch releases, enabling users to validate patched versions with their own workloads by (re-)recording with the old version and replaying with the new version.
- An open-source prototype implementation, portable across Linux distributions running on x86-64, of ad hoc test generation for security vulnerabilities and other critical bugs.
- A novel collection of buggy and fixed program versions and simplified exploit execution scripts with an emphasis on CVE vulnerabilities.

2 ARCHITECTURE AND DESIGN

Our ad hoc test generation workflow constitutes four main procedures: recording, static preprocessing, load time quilting and the runtime decision making algorithm, which we describe in turn. Recording and the two preparation stages are shown in Figure 3, with the resulting runtime depicted later on in Figure 11.
They remain unchanged even if their addresses and references change. After processing the patchfile we use the symbol tables and global variables. Since symbols are the points of reference between original and patched binaries, it must maintain the binary context of the original binary as illustrated in Figure 7.

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 (e.g., 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 point of reference between the old and the modified binaries. They remain unchanged even if their addresses and references change. After processing the patchfile 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 5 and 6 for concrete examples.

Most real world projects create multiple binaries and associated libraries when building 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.

Pre-Load Steps for Quilting. Once the function and line addresses have been resolved via the procedure described above, 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 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 illustrated in Figure 7.

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 (e.g., 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.

2.3 Load Time Quilting

Loading Replication & Custom ATTUNE 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) cannot be resolved until after loading completes. So even though ATTUNE knows which references need resolution pre-load, it can’t actually resolve those references until load time. In order to preserve the integrity of the replay, all required shared libraries, executables, and system libraries must be loaded into the recorded memory locations. Shared libraries and executables required for replay are included in the trace, 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) which hijacks execution from the system loader and begins the replay process. As mentioned earlier, 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 7 depicts the address space when loading has completed.

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 8, and consists of both pre-load and load-time actions. The process starts from the address of the modified function as determined from the patchfile and DWARF processing. The modified function is scanned for references. When a reference is identified, if the pointer is effected by the quilting process then ATTUNE’s translation procedure corrects the pointer. The log messages in 9 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.

Figure 7: Address Space Detail

Figure 8: Pointer Translation Procedure

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: 0xaa60 // target address in the original binary

Resolving string reference at: 0x1b214 //patch references string "chunk data is too large"
Resolving offset ... for "chunk data is too large"
Found string: "chunk data is too large" at 0x120d //identified string in original binary
... module pngutil code found at 0x000000
... module pngutil data found at 0x200000
... generating quilted code

Figure 9: libpng-bug-1 abbreviated linking example

PCT Code, PLT Entries & Trampolines. Position independent code compilation has become the standard for security and efficiency reasons so modern binaries can be loaded anywhere in the address space. As a result the locations of external functions and symbols aren’t 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 only resolved 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 for our purposes we can’t rely on a PLT because the system loader which 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. An example transformation is in Figure 10.

Resolving String & Data Sections. Other than references to code sections the patched code may 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
The compiler accesses data through a global offset table entry, but inform decision making. For any non-deterministic event that takes code where we use information about added or deleted lines to function is called. We break at that point and move to the patched have been modified and perform a strict replay until a modified continue to leverage developer environment information to aid code generation time it knows where the data has been loaded. PLT[n]: jmp *GOT[n] prepare resolver jmp PLT[0] Figure 10: PLT Transformation PLT[printf]: mov ebp, esp mov esp, %ebp jmp printf ... Figure 11: Runtime Architecture symbols and strings function as a point of reference between the modified and the original binary. These translations can be done as described in Figure 8, 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. To accomplish this another small transformation must take place. The compiler accesses data through a global offset table entry, but cannot use it because the global offset table was compiled for the modified code. Instead ATTUNE points to the data directly since at code generation time it knows where the data has been loaded. 2.4 Runtime Replay Decisions The runtime architecture is shown in Figure 11. 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 Figure 12. 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 the 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 appropriate 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 which 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 IO such as any operation involving a file descriptor including network or device IO are harder to replay since they require a specific kernel state. We have to actually recreate the file state as best we can 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 when to deliver signals based on the value of the retired conditional branches (RCB) performance counter standard in Intel chips. For signals that have been recorded based on signal type, we check if we are in an inserted line. If we are then we deliver the signal and assume it’s created by the patch (e.g. 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.

2.5 Limitations ATTUNE relies on rr as-is for the original recording (re-recording from some lightweight tracing mechanism) and for replay of that recording with the original version of the program [48–50].
ATTUNE nor rr provide any special support, beyond interactive debugging (rr integrates with gdb) that helps developers locate and understand the root cause of the bug sufficiently to develop successful patches [34, 36, 63]. ATTUNE extends record-replay to test those patches as if they had been in place in the user environment where the security vulnerability was discovered.

Our ATTUNE prototype inherits the limitations of rr. Most notably, rr runs the recording on a single thread during replay, so replayed parallel programs incur the slowdown of a single core [49, 50]. ATTUNE accommodates thread synchronization, and faithfully emulates the error state, but because rr simulates thread interleavings by interrupting a single thread execution, ATTUNE cannot accurately verify patches for concurrency vulnerabilities. ATTUNE also does not support significant changes to data structures, e.g., changing the size of a struct on the stack or in the heap. ATTUNE does not verify patches to preprocessor macros. Since macros are inserted inline when executables are generated; there are no associated symbols so a macro cannot be replaced in the same way that ATTUNE replaces functions. In principle, ATTUNE could be augmented to instead replace each function that calls a changed macro, an engineering effort.

3 EVALUATION

We evaluated ATTUNE on a Dell OptiPlex7040 with Intel Corei7-6700 CPU at 3.4GHz with 32GB memory, running Ubuntu 18.04 64-bit, using gcc/g++ version 7.4.0 and python 3.4.7 (which come standard with Ubuntu). ATTUNE is built using the CMake scripting language version 3.10.2 and Make version 4.1. Since we are not aware of any published benchmarks for Linux/C patch testing, we collected security vulnerabilities and other significant bugs discovered and patched 2016–2019 from popular open-source projects, and recorded corresponding scenarios and exploits. Upon paper acceptance, we will provide Ubuntu scripts to recreate all the studied executions together with ATTUNE’s open-source implementation. For brevity we present only a few in detail.

Our evaluation answers three primary research questions (RQs):

- RQ1: What types of patches can ATTUNE successfully accommodate and what are its limitations?
- RQ2: What are the constraints ATTUNE places on the developer’s patching efforts?
- RQ3: What are the constraints ATTUNE places on user validation efforts?

RQ1: What types of patches can ATTUNE successfully accommodate and what are its limitations?

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 13 [1] 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. (The code shown in our figures is abbreviated.) Since ATTUNE replaces the entire modified function instead of individual lines of code, it needs to resolve all references in the new version, e.g., to string manipulation functions.

ATTUNE recreates the input 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 developer can set a breakpoint at the patched section, watch the if statement process the input correctly and verify the string in "portptr". Since the log has no information regarding how the network would have responded to the http request had it been sent, the test ends. This is a very simple case.

Figure 14 [2] 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 scanf 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, an attacker could craft a malicious PNG image that triggers a bad calculation of row_factor in Figure 15 [7], causing a divide-by-zero error and Denial-of-Service. With traditional bug reports, the user sends the image as an attachment — assuming, of course, they are aware of this 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, in contrast to the original error handling.

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. While not particularly strenuous
There is no need to send a file with the problematic non-standard conventional bug reports as files in transit may arrive with modified allows monitor connections to send logging and status checking test and debug using conventional mocks, as complex network one such example in redis saw involved adding conditionals. Many security-critical patches can verify the patch by letting the program run to termination and diverge drastically from the original execution trace. The developer wc ing any additional information and successfully returns from the patched versions of the new function and those functions that call a challenge from A A patch for the file processing utility adds special charac- ter parsing functions as shown in Figure 16 [10]. While the change is small, the function (used only in one place) is prohibitively long it will significantly hinder a developers ability to 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 18 [12] 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.

**Miscellaneous and Limitations** ATTUNE successfully constructed test cases in scenarios that swapped library function calls [13] and swapped control flow blocks in [5]. Applying ATTUNE to parameter changing in [4] was unsuccessful, but this appears to be an engineering problem not an approach limitation. ATTUNE failed to locate pieces of the modified function in the loaded binaries and couldn’t link a patch. This is likely caused by build idiosyncrasies that don’t include the buggy filename in the DWARF information. ATTUNE depends on DWARF information for line numbers so test construction was unsuccessful.

A removed break statement in [10] caused a surprising error. While the change is small, the function (used only in one place) is inlined, so there is no symbol table entry. ATTUNE would accommodate in-lining with additional engineering effort.

**Figure 16: wc-bug-1 New Function and Refactoring**

```c
void addReplyErrMsgLength(const char *msg,
    const char *error,...)
{
    if (c->flags & TTUNE_HOST) {
        if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)) {
            if (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)) {
                goto word_separator;
            } else {
                goto word_separator;
            }
        } else {
            goto word_separator;
        }
    } else {
        if (isisspace (c->flags & (CLIENT_MASTER|CLIENT_SLAVE)) {
            goto word_separator;
        } else {
            goto word_separator;
        }
    }
}
```

**Figure 17: redis-bug-1 Erroneous Conditional**

```c
url_parse (const char *url,...) { 
    /* check for invalid control characters in host name */
    for (p = url; *p; p++) {
        if (c->flags & TTUNE_HOST) {
            url_free(url);
            error_code = PE_INVALID_HOST_NAME;
            goto error;
        }
    }
}
```

**Figure 18: wget-bug-2 New Loop**

```
from the developer’s perspective, these types of fixes do create a challenge from ATTUNE’s perspective. 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 functions as shown in Figure 16 [11]. 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 without requiring any additional information and successfully returns from the modified functions displaying the patched output. Since we largely contains deterministic operations, testing the modified 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 17 [9] 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.

**RQ1 Answer:** ATTUNE successfully generates ad hoc test cases for patches involving string parsing, mathematical operations, new functions and parameter refactoring, adding conditionals, new/changing loop conditionals, changing library calls, and swapping control-flow blocks.

**RQ2: What are the constraints ATTUNE places on the developer’s patching efforts?**

**Quilting Overhead** Since ATTUNE’s quilting occurs at load time, the procedure occurs on every test run. ATTUNE parses all the binaries for pertinent information at the start of each run, and reconstructs the patched code each time. If this procedure is prohibitively long it will significantly hinder a developers ability to produce a patch. Figure 19 shows our measurements of recording overhead. In the worst case the overhead incurred is slightly below 4 seconds and in the best case near instantaneous. Overhead incurred by the quilting procedure depends on the time it takes to parse the binaries, search for pointers, and update those pointers. The majority of the overhead is incurred by parsing the binaries since
While most patches are rather small in size, our evaluation showed
A
the worst case overhead was close to 100 Kilobytes in [3], but otherwise tended to remain under 25 KB. Modern Linux Systems with 64bit address spaces and 4KB page sizes can easily accommodate this overhead. The memory footprint is not related to the size of the patch but instead is related to the size of the functions which have been patched and the associated data those functions access. In [3] the patch spanned multiple large functions. As such memory overhead is tied both to the nature of the patch and the nature of the code base. If the code base contains large monolithic functions, then patching those functions will incur higher overhead, and the more functions the patch spans the higher the overhead will be.

Patch Sizes. We want ATTUNE to remain as transparent to the developer as possible so the developer can work unimpeded. While most patches are rather small in size, our evaluation showed that we could accommodate rather large code changes as well (Figure 21). It is important to note that test construction is built from the recorded events, so even though the developer may make far reaching changes the coverage of the test case is not related to the changes the developer makes. In the case of [13] which makes far reaching changes across the code base to address the same bug in multiple places, ATTUNE allows for major changes, but the test case constructed actually only covers a small subset of the patch. ATTUNE constructs the test case to cover the minimal patch, but if the developer decides to make additional changes they must construct additional test cases by hand.

RQ2 Answer: ATTUNE successfully generates ad hoc test cases quickly and efficiently even for relatively large code changes, but requires the developer to write additional tests for patches that fix additional bugs outside available user execution traces.

RQ3: What are the constraints ATTUNE places on user validation efforts?

Recording Overhead. ATTUNE uses a verbose recording system to guarantee bug reproduction. As such it is not practical for production recording and instead should be used for bug capture in an offline setting as outlined in Figure 2. Figure 22 shows the high overhead incurred during the recording procedure. In the worst case showing almost 60x performance and usually presenting a 10-20x slowdown. [49] contains complete performance metrics that are consistent with our findings.

Execution Trace Sizes. Our ad hoc test generation workflow assumes that users send execution traces to developers for reproduction and debugging, so we do not want the log files from re-recording to become too large and cumbersome. Figure 23 shows exact measurements. We expected a dependency between log sizes and the number of events recorded, but found that they are largely independent. This is because the files used in the execution trace...
3.1 Threats to Validity

**Internal.** As far as we know, neither rr nor any other record-replay system was recording execution traces when any of the real bugs we studied were discovered. Some of our scripts for recording the buggy version run bug reproduction tests included in the real bug reports, but others were contrived. This threat is partially mitigated since the contrived scenarios were developed by a three masters students who were not ATTUNE developers. We describe how we imagine a developer would verify the 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. Lastly, since do not have execution traces for any real users using the programs in our dataset, we simulated workloads with benchmarks that may not be representative of how real users would validate these programs.

**External.** We demonstrate that ATTUNE supports a wide variety of single-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.

4 RELATED WORK

Kuchta et al [39] generates tests for software patches using “shadow symbolic execution”. The old and new program versions are symbolically executed in tandem, with the old version shadowing the new one. 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.

Elbaum et al [26] 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., [40], with analogous advantages and disadvantages.

Kravets and Tsafir [38] proposed “mutable replay”, a hypotheti-
cal design to construct a new execution trace for a modified pro-
gram from an execution trace of a previous program version that, as in ATTUNE, demonstrates a bug. Mutable replay was later implemented by Viennot et al [62] building on the Scribe record-replay system [42]. This leveraged checkpoint/restart [41] in a backtracking search algorithm that sought to minimize adds/deletes to the
original execution trace. Although successful on many bug-fix examples in the sense that execution continued through the modified code, the minimal-distance execution trace is not necessarily the same as would have occurred had the modified code been running in the user environment, which is what ATTUNE aims. Scribe required a shared file system (copy on write) between the user and developer environments and a special Linux kernel module that intercepted and controlled system calls and other non-deterministic kernel events within both user and developer environments, which are impractical for most post-deployment scenarios, whereas ATTUNE runs without privileges in user-space with no changes to the operating system and no sharing between user and developer environments other than user-submitted execution traces.

Parallel retro-logging allows developers to change their logging instrumentation and then quickly see what the new logging would have produced on a previous execution [55], but the program itself is not modified. Arora et al [14] describe feeding cloned network traffic to a sandboxed fork of an architectural component in a service-oriented architecture, for debugging or testing patches of that component, but the sandboxed execution trace is not necessarily faithful when there are non-network sources of non-determinism.

There are numerous other record-replay tools in the literature, recently including [44, 45, 54, 58, 68]. Some versions of gdb build-in recording and replaying debugging sessions [28], as does Microsoft’s IntelliTrace [47]. These tools reproduce execution traces for a given program version and cannot test modified versions.

Many record-replay tools focus on reproducing concurrency bugs, e.g., [18, 33, 56], outside the scope of this paper. 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 record-replay research focuses on reducing the overhead of recording, e.g., [32, 35, 51]. Cui et al [24] explain that “high-fidelity program tracing is not affordable in deployed systems”, so their REPT tool combines hardware tracing and binary analysis to reconstruct execution traces, which can then be replayed with the same program version. Castor [46] records multi-core applications by leveraging hardware-optimized logging, transactional memory, and a custom compiler. It can replay slightly modified binaries when the changes do not impact program state. Pervasive (always-on) recording will likely require special hardware, operating system and/or compiler support for the foreseeable future. ATTUNE users can choose any baseline recording tool that supports faithful replay. Only execution traces known or suspected to contain evidence of security vulnerabilities or other bugs need to be replayed by the host recording system for ATTUNE’s offline re-recording.

Multi-Version Execution (MVE) provides an alternative approach to user validation. ATTUNE’s validation of patched programs in the user environment proceeds by: lightweight recording of the user’s production workloads with the old version of the program, offline re-recording, replaying ad hoc tests generated from those workloads on the patched version and, if all is satisfactory, switching the new version into production by some mechanism outside ATTUNE, e.g., “mutable checkpoint-restart” [29]. In MVE, the patched and original versions run simultaneously on production user workloads, adding runtime overhead but enabling immediate detection of undesirable divergences [31, 52].

Fuzzing seeks inputs that induce crashes and other problems [53]. Other approaches also strive to induce bad behaviors, e.g., [15, 66]. [59] builds on EvoSuite’s search-based testing [57] to reproduce crashes. Symbolic execution [17] 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 [43]. These kinds of tools, as well as conventional regression testing, are complementary to ATTUNE.

5 CONCLUSION

ATTUNE (Ad hoc Test generation ThroUgh biNary reWriTing) supports ad hoc test generation for security vulnerabilities and other critical bugs discovered post-deployment, when there are no existing developer tests for bug reproduction and testing candidate patches, and little time for constructing and vetting new developer tests. ATTUNE first qtyulates the modified functions (the patch) into the original binary and then interprets the execution trace from the original binary, as it executed in the user environment, to emulate the generated ad hoc test on the patched binary. The developer just modifies one or more buggy functions to produce a candidate patch and monitors the progress of the ad hoc test to check that the bug no longer manifests; the developer does not intervene in ATTUNE’s binary rewriting and testing and does not need to build test scaffolding. ATTUNE also produces metadata that the developer can deploy with the patched program, which enables users to validate the new version by using ATTUNE to (re-)record execution traces of their own workloads with the original version and emulate the corresponding ad hoc tests with this new version. We showed that ATTUNE generates ad hoc tests for a wide range of known security vulnerabilities and bugs in older versions of open-source software, with minimal developer effort. We will release ATTUNE and our dataset open-source on github upon acceptance of this paper.
REFERENCES


Acknowledgments

This work was supported by the National Science Foundation under grant CCF-1855343. We would like to thank the anonymous reviewers for their constructive feedback. We also thank the participants in the Software Engineering and Security (SE-Sec) workshop for their valuable insights. Finally, we are grateful to the authors of the software projects used in our experiments for their contributions to open-source software.

11