Ad hoc Test Generation Through Binary Rewriting

Anthony Saieva
Department of Computer Science
Columbia University
New York NY, USA
ant@cs.columbia.edu

Shirish Singh
Department of Computer Science
Columbia University
New York NY, USA
shirish@cs.columbia.edu

Gail Kaiser
Department of Computer Science
Columbia University
New York NY, USA
kaiser@cs.columbia.edu

***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 attackers pounce on exposed user installations. This can be challenging when factors in a specific user environment triggered the bug. If enabled, however, 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. Many modern record-replay tools integrate interactive debuggers, to help locate the root cause, but don't help the developers test whether their patch indeed eliminates the bug *under those same conditions*. In particular, modern record-replay tools that reproduce intermediate program state cannot replay recordings made with one version of a program using a different version of the program where the differences affect program state.**

This work builds on record-replay and binary rewriting to automatically generate and run targeted tests for candidate patches significantly faster and more efficiently than traditional test suite generation techniques like symbolic execution. These tests reflect the arbitrary (ad hoc) user and system circumstances that uncovered the bug, enabling developers to check whether a patch indeed fixes that bug. The tests essentially replay recordings made with one version of a program using a different version of the program, even when the the differences impact program state, by manipulating both the binary executable and the recorded log to result in an execution consistent with what would have happened had the the patched version executed in the user environment under the same conditions where the bug manifested with the original version. Our approach also enables users to make new recordings of their 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 break functionality they rely on.

***Index Terms*—test generation, software patching, record-replay, binary rewriting, security vulnerabilities**

I. Introduction

Developer testing may not be representative of how software is used in the field [1]. User bug reports [2], [3] and vulnerability reports [4], [5] are populated primarily with bugs that were not discovered by developer tests. But reproduction steps included in bug reports are often insufficient to reproduce the bug [6]. Thus when a security vulnerability or other critical bug is not detected by developer testing prior to deployment, but reported by users, developers need to construct a new test that both reproduces the bug in the original version of the code and verifies the absence of the bug in the patched code.

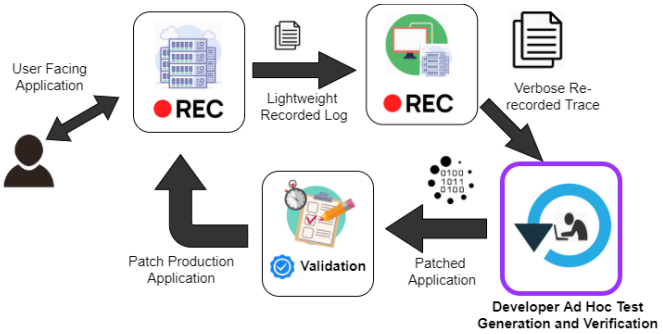

Fig. 1. Ad hoc Test Generation Concept

This paper presents a novel approach for rapidly generating tests that reproduce post-deployment bugs, support debugging, and verify that candidate patches do not exhibit the buggy behavior. We build on state-of-the-art record-replay technology that records execution traces in the user environment, reproduces post-deployment bugs, and supports debugging, but state-of-the-art record-replay tools do not support testing candidate patches. Since bug-fix patches may introduce other errors not detected by developer tests, breaking users' mission-critical functionality [7], our approach also supports user validation of released patches.

The gist of our patch-testing solution is illustrated abstractly in Figure 1. Production applications are recorded by some always-on lightweight recording mechanism. When a bug is discovered, the lightweight recording is replayed and re-recorded offline to augment with additional context necessary for replaying in a different environment. The verbose log is shipped to the developer, who writes and verifies prospective patches. Finally, the new release is accompanied by patch-specific metadata enabling users to validate the patch.

We refer to this concept as *ad hoc test generation* because the generated test emulates whatever user context manifested the bug. We emphasize that ad hoc test generation is intended only for urgent time-crunch situations, when there are no existing developer tests that detect the bug and careful planning and design of new developer tests would take too long. Ad

The Programming Systems Laboratory is supported in part by NSF CNS-1563555, CCF-1815494 and CNS-1842456.

hoc test generation is feasible when execution trace divergence is small, analogous to Tucek et al's "delta execution" [8], 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 changing core program semantics, shared memory layout or process/thread layout.

The premise of record-replay technology is that there are behaviors that manifest in the user environment that cannot be reproduced by simply running the program with known inputs in the developer environment. If there are such known inputs, ad hoc testing is easy – just run the program with those inputs. This paper addresses more complicated scenarios.

We have developed a novel ad hoc test generation tool, ATTUNE (**A**d hoc **T**est generation **T**hro**U**gh bi**N**ary r**E**writing). 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 emulates the original execution context, including external inputs, environment variables, the results of system calls, network connections, and the accessed portions of the file system, databases and other local resources as they were at the time the exploit or bug manifestation. Unlike existing record-replay tools, ATTUNE leverages binary rewriting [9] to modify the original executable at load-time to insert the patched functions from the modified executable, and then interprets the recorded log to manipulate the test emulation as it executes the patched functions. Inserting the patched functions into the original binary results in an execution that perfectly matches the recorded log until divergence when the first patched function is reached. Continuing the execution beyond this point enables the developer to assess whether a candidate patch indeed fixes the bug.

ATTUNE is based on two key insights: The first is that the symbol tables in Linux ELF files provide points of reference between original and patched versions. Thus each patched function can replace the corresponding original function and access the same global variables and strings. Our second key insight is the recorded log of the original execution trace does not need to be replayed verbatim in order. Instead, events in the log can be skipped or swapped, and new events can be derived on the fly from those in the log, to match the patch.

ATTUNE follows the workflow illustrated in Figure 1 to generate an ad hoc test for candidate patches faithful to the execution trace recorded in the user environment. 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 [10]. Then when user observation, analysis, monitoring, etc. determines that a lightweight trace manifests a security vulnerability or other bug, then that trace is augmented offline by ATTUNE's verbose recorder.

Our ATTUNE prototype builds on Mozilla's **rr** open-source record-replay tool [11]–[13] as the verbose recorder. ATTUNE like rr runs without privileges in user-space, with conventional hardware, operating system, compiler, libraries, build pro-

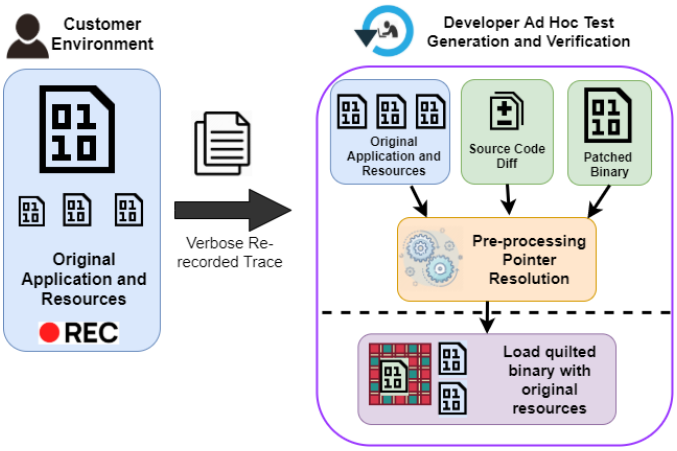

Fig. 2. Recording and Preparation for Ad Hoc Test Generation

cesses etc. and no changes to the application. This represents a stark contrast to other test generation techniques like symbolic or concolic execution. ATTUNE's binary rewriting modifies the application executable only at load-time, i.e., in memory, not the executable file(s). While the technical details of our binary rewriting mechanisms are specific to our modification of rr's replayer, ad hoc test generation is not, and in principle ATTUNE prototypes could be built on any record-replay technology that supports sufficiently detailed execution traces.

We explain our requirements for verbose execution traces and the technical details of our binary rewriting techniques in Section II. Our evaluation in Section III describes how a developer would use ATTUNE to test candidate patches for a variety of bugs from well-known open-source projects. Section III 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. Finally, our evaluation compares to the time and space costs of using KLEE [14] to generate a test that covers the patched part of the code, given that the bug has already been found and its location known. Section IV discusses related work.

The contributions of this paper are:

- An approach to leveraging record-replay technology and binary rewriting to generate ad hoc test cases to exercise candidate patches as if they had been executing in the user context, 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, available at https://github.com/Programming-Systems-Lab/ATTUNE.

II. Architecture and Design

Our ad hoc test generation workflow has four main components: *recording*, *static preprocessing*, *loadtime quilting* and

```
--- 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
```

Fig. 3. libpng-1 Abbreviated Example Patchfile

the *runtime replay decisions*. Recording and the two preparation stages are shown in Figure 2, with runtime depicted in Figure 10. Both preparation stages leverage the open-source Egalito recompilation framework [15].

A. 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 as in Figure 1. 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.

B. Static Preprocessing

Source Code and Binary Preprocessing. Figure 3 shows an abbreviated example patchfile from a libpng bug-fix [16]. Patchfiles document which files changed, which function in the file changed, and which lines within that function were inserted and deleted. Patchfiles 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
```

Fig. 4. 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
```

Fig. 5. 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 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 4 and 5 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.

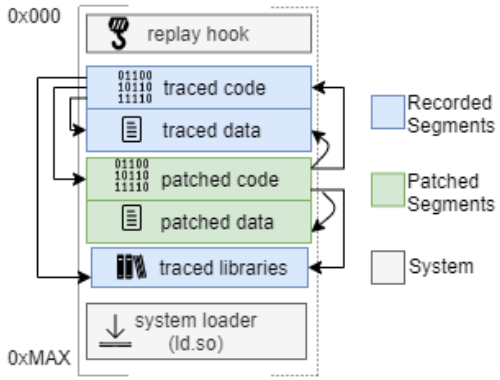

Fig. 6. Address Space Detail

C. 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 6.

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

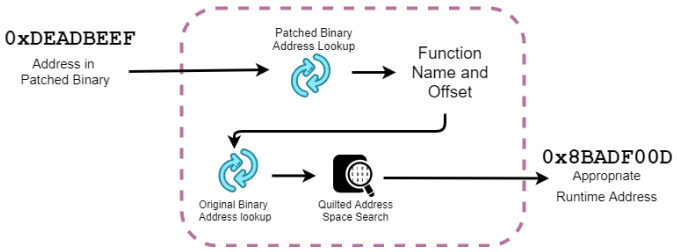

Fig. 7. Pointer Translation Procedure

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. So even though ATTUNE knows pre-load which references need resolution, 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 6 depicts the address space when loading has completed, and Algorithm 1 outlines the loading 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 7,

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
```

```
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
```

Fig. 8. libpng-1 abbreviated linking example

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 affected by the quilting process then ATTUNE's translation procedure corrects the pointer.

The log messages in Figure 8 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 refer-

Fig. 9. PLT Transformation

ences, 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 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 9.

Resolving String & Data Sections. The patched code may also reference data section variables like global data and strings. The patched code must reference the old code where possible and the patched code where required. Identical symbols and strings function act as points of reference between the modified and the original binary.

These translations are similar to Figure 7, 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.

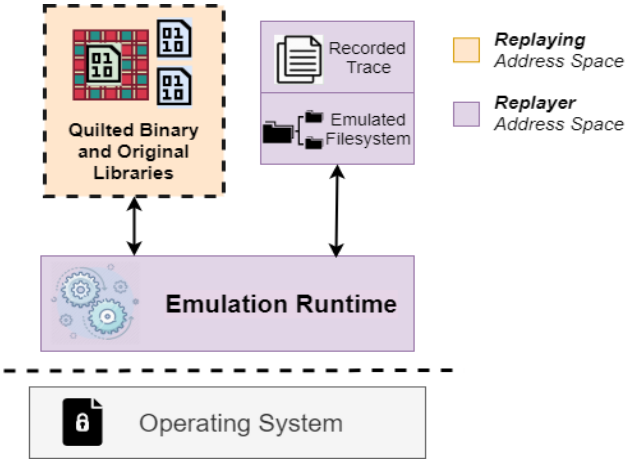

Fig. 10. Runtime Architecture

D. Runtime Replay Decisions

The runtime architecture is shown in Figure 10. 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 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, 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 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.

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
```

III. 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. For comparisons with KLEE [37], we used KLEE version 2.1 [38] (the most recent version as of this writing) compiled with LLVM 9.0.1.

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 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 11.

We empirically compared ATTUNE with KLEE [14], a state of the art test suite generation tool. We limited our comparison to bugs in Table 11 that are part of the coreutils project since KLEE supports coreutils easily. Our other studied bugs have more external libraries, aside of libc, so require additional engineering for KLEE to accommodate. KLEE was given 60 minutes before a timeout ended the test generation, as in [37].

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

ATTUNE successfully validated the real developer patches for 19 and failed for 2 of the bugs the challenge team collected, marked with ✓and ✗ in Table 11, 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 12 [17] 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 developer 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 13 [25] 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*

Bug	Success or Failure	Patching Effort	Files Modified	LOC Changed
curl-1 [17]	✓	Changes how a string is parsed	1	16+, 16-
curl-2 [18]	✓	Changes functions arguments and call.	4	9+, 9-
curl-5 [19]	✓	Modified if statement for buffer overflow	1	4+, 1-
curl-6 [20]	✓	Added new function and inserted call	1	55+, 8-
curl-8 [21]	✗	Changes multiple functions calling a write function	4	17+, 17-
curl-9 [22]	✓	Change parameters to a function call	1	2+,1-
curl-10 [23]	✓	Adds a condition check	1	6+, 2-
curl-11 [24]	✓	Off by 1 correction	1	1+, 1-
curl-12 [25]	✓	Changes libc calls to add extra parsing	1	5+, 5-
libpng-1 [16]	✓	Calculation modification for divide by 0 error	1	6+, 3-
libpng-2 [26]	✓	Adjust calculation for idat chunk max	3	13+, 13-
wc-1 [27]	✓	Added new function and changed condition check	1	23+, 2-
wc-2 [28]	✓	Added error condition check	1	3+, 0-
yes-1 [29]	✓	Substantial changes in option parsing	15	40+, 141-
shred-1 [30]	✗	Removed a break statement	1	1+, 1-
ls-1 [31]	✓	Added condition for change in option parsing	1	1+, 2-
mv-1 [32]	✓	Adding a conditional check before operation	2	6+, 0-
df-1 [33]	✓	Replacing open calls with stat calls	2	12+,8-
bs-1 [34]	✓	Changing a loop condition	1	2+, 1-
wget-1 [35]	✓	Adding conditional check for log	1	1-, 2+
redis-1 [36]	✓	Adding conditional check	1	1+, 1-

Fig. 11. Patch-Testing Dataset

```
    ...
+ 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;
...
```

Fig. 12. Curl-1 URL Parsing

```
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) {
     ....
```

Fig. 13. Curl-12 String Parsing

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 14 [16], 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 unlikely to be aware of the carefully crafted malicious image uploaded by an attacker. ATTUNE does not require attachments besides the execution trace, since

```
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);
```

Fig. 14. 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;
}
...
```

Fig. 15. wc-1 New Function and Refactoring

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 from the developer's perspective, these types of fixes do

```
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";
...
```

Fig. 16. 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;
+   }
+  }
```

Fig. 17. wget-2 New Loop

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 15 [27]. 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 16 [36] 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 17 [39] 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 [29] and swapped control flow blocks df-1 [33]. 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 challenge 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 [30] 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 [21] 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.

As Table 18 shows, ATTUNE created tests for 7 of the 8 studied coreutils bugs, while KLEE created tests for 5 and timed out on 2 that were handled by ATTUNE (wc-2, bs-1). Neither was able to generate tests for the shred-1 bug.

B. ATTUNE*'s wait time and memory overhead is small*

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 searching the program space is removed. Table 18 shows our measurements of quilting overhead relative to KLEE. In all but one case our speedup was well over 90% and could reduce generation time by as much as 99.6%. In absolute time measurements our worst case was just under 4 seconds.

Memory Footprint: ATTUNE inserts patched code during testing so it incurs some memory overhead at test time.

Bug	ATTUNE	KLEE	ATTUNE Time	KLEE Time	Speedup	ATTUNE Mem	KLEE Mem	Overhead Cut
wc-1 [27]	✓	✓	1.37s	300.046s (5m)	**99.5%**	5.9 KB	108.388 KB	**94.5%**
wc-2 [28]	✓	✗	1.277s	na	na	2.8 KB	107.7 KB	**97.4%**
yes-1 [29]	✓	✓	3.4s	8.569s	**60.3%**	10.6 KB	107.09 KB	**90.1%**
shred-1 [30]	✗	✗	na	na	na	na	na	na
ls-1 [31]	✓	✓	1.6s	19.57s	**91.82%**	7.4 KB	132.9 KB	**94.4%**
mv-1 [32]	✓	✓	3.6s	58.4s	**93.84%**	4.3 KB	208.2 KB	**97%**
df-1 [33]	✓	✓	1.48s	18.869s	**92.15%**	5.97 KB	151 KB	**96.05%**
bs-1 [34]	✓	✗	1.2s	na	na	5.6 KB	113.37 KB	**95.06%**

Fig. 18. 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
...
```

Fig. 19. redis-bug-1 Metadata for User Validation

However, thanks again to targeted testing this is a one time operation. 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%.

C. Users validate released patches with their own workloads

In the last (optional) stage of the patching workflow, the user validates the patch in their own environment to verify no needed functionality has broken. This stage would be particularly useful if combined with our separate static quilting tool [40], which enables users to select individual bug-fix patches from new releases containing other unrelated changes. Because ATTUNE operates entirely in user-space, without hardware, operating system, etc. support, it can run in both developer and user environments. ATTUNE summarizes the "diffs" in source and binary code, and exports metadata along with the released binary patch allowing for user validation.

For sample user environment workloads, we used the redis benchmark [41], which simulates thousands of different requests to the server, and the *httperf* benchmarking tool [42] making thousands of connections. The validation procedure for the redis patch [41] is similar to the redis discussion above, but ATTUNE utilizes only the metadata it added to the released patch, shown in Figure 19.

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 mapped into memory.

D. Threats to Validity

Internal. As far as we know, no execution traces were recorded when any of the studied bugs 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 three grad students who were not ATTUNE developers. We describe how we imagine a developer would verify 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.

E. 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 [11], [12]. 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 [11]–[13]. Since rr was designed to be used during developer testing, with too high overhead for production [11], we adopt the re-recording model shown in Figure 2. 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 [10] explain this limitation is generally unimportant in practice.

A few ATTUNE limitations are orthogonal to the 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.

IV. Related Work

iFixR [2] 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 [43] 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. [44] 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 [45] combines symbolic execution with heuristics to generate test cases that cover the patched part of the code, while shadow symbol execution [46] 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 [47] allows developers to change their logging instrumentation so previous executions produce augmented logs, but the program is not modified. Parikshan [48] feeds cloned network traffic to a sandboxed component of a service-oriented architecture, for debugging or testing patches of that component, but the sandboxed execution is not faithful for non-network sources of non-determinism.

Kravets and Tsafrir [49] proposed "mutable replay", a hypothetical design to execute a patched program reusing the system calls and other non-deterministic events from a recorded execution trace, as in ATTUNE. Mutable replay was later implemented in Dora [50], building on the Scribe record-replay system [51]. Dora leveraged checkpoint/restart [52] in a backtracking search to minimize adds/deletes to the recorded execution trace. Although successful on many bug-fix examples in the sense that execution continued through the patched part of the code, the minimal-distance execution is not necessarily the same as would have occurred had the patched code been running in the user environment, as ATTUNE aims. Dora and Scribe relied on a 2.6.x kernel module that intercepted and controlled system calls and other events within both user and developer environments. This module will not operate in modern Linux kernels. In contrast, ATTUNE runs without privileges in user-space with no special OS support.

There are numerous other record-replay tools in the literature, e.g., [53]–[59]. These tools reproduce executions, but none tests patched versions. Much research focuses on reducing recording overhead, e.g., [60]–[62]. Cui et al [63] combines hardware tracing and binary analysis to reconstruct execution traces, which can then be replayed with the same program version. Castor [10] records multi-core applications by leveraging hardware-optimized logging, transactional memory, and a custom compiler. It can replay slightly modified binaries that do not impact program state.

Multi-Version Execution (MVE) provides an alternative approach to user validation. We envision that the user records production workloads with the old version and re-records offline as in Figure 1, 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., mutable checkpoint-restart [64]. In MVE, the patched and original versions run simultaneously on production user workloads, adding runtime overhead but enabling immediate detection of undesirable divergences [65], [66].

V. 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. ATTUNE also produces metadata that the developer can deploy with the patched program, which enables users to validate the patch by using ATTUNE to create additional ad hoc tests from their own workloads. 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. Our open-source implementation is available at https://github.com/Programming-Systems-Lab/ATTUNE.

VI. Acknowledgment

We thank Yangruibo Ding, Victor Xu, and Ziao Wang for compiling the patch-testing dataset. We thank Robert O'Callahan, the lead rr developer, and David Williams-King for their feedback, and thank Jason Nieh for his input.

REFERENCES

[1] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?" in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 321–332. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST.2017.36

[2] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and 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. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338935

[3] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300536

[4] Cybersecrity & Infrastructure Security Agency, "CISA coordinated vulnerability disclosure (CVD) process," https://www.cisa.gov/coordinated-vulnerability-disclosure-process, December 2019.

[5] hackerone, "Vulnerability disclosure philosophy," https://www.hackerone.com/disclosure-guidelines, July 2019.

[6] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and 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. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338947

[7] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 26—-36. [Online]. Available: https://doi.org/10.1145/2025113.2025121

[8] J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with Delta Execution," in *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508267

[9] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys*, vol. 52, no. 3, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3316415

[10] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and 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. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037751

[11] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *USENIX Annual Technical Conference (USENIX ATC)*, July 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[12] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability: Extended technical report," *arXiv:1705.05937 [cs.PL]*, May 2017. [Online]. Available: http://arxiv.org/abs/1705.05937

[13] Mozilla, "what rr does." [Online]. Available: https://rr-project.org/

[14] KLEE Team, "KLEE LLVM execution engine," http://klee.github.io/, last accessed 8/11/20.

[15] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2020, pp. 133—-147. [Online]. Available: https://doi.org/10.1145/3373376.3378470

[16] Red Hat Bugzilla – Bug 1599943, "libpng: Integer overflow and resultant divide-by-zero," https://bugzilla.redhat.com/show_bug.cgi?id=1599943, 2019, CVE: https://nvd.nist.gov/vuln/detail/CVE-2018-13785.

[17] "Curl string parsing bug," https://github.com/curl/curl/pull/3365, 2019.

[18] "Curl string parsing bug," https://github.com/curl/curl/pull/3381, 2019.

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

[20] "Curl security vulnerability," https://github.com/curl/curl/pull/3433, 2019.

[21] "Curl change parameter fix," https://github.com/curl/curl/commit/-e50a2002, 2019.

[22] "Curl_follow: accept non-supported schemes for "fake" redirects," https://github.com/curl/curl/commit/-2c5ec339ea67f43ac370ae77636a0f915cc5fbeb, 2018.

[23] "URL: fix IPv6 numeral address parser," https://github.com/curl/curl/pull/3219, 2018.

[24] "Curl globbing error," https://github.com/curl/curl/issues/3251 , 2019.

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

[26] "libpng IDAT miscalculation," https://sourceforge.net/p/libpng/bugs/270/ , 2019.

[27] "wc special character bug," https://github.com/coreutils/coreutils/commit/-a5202bd58531923e, 2019.

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

[29] "yes coreutils library function," https://github.com/coreutils/coreutils/-commit/44af84263e, 2019.

[30] "shred coreutils library function," https://github.com/coreutils/coreutils/-commit/c34f8d5c787e6, 2019.

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

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

[33] "df coreutils library function," https://github.com/coreutils/coreutils/-commit/b04ce61958c, 2019.

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

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

[36] "Redis monitor request causes crash," https://github.com/antirez/redis/commit/e2c1f80b, 2019.

[37] C. Cadar, D. Dunbar, and 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.

[38] KLEE Team, "Stable releases of KLEE," http://klee.github.io/releases/, last accessed 8/11/20.

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

[40] A. Saieva and G. Kaiser, "Binary quilting to generate patched executables without compilation," in *Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, November 2020, in press.

[41] "Redis benchmark tests server functionality," https://github.com/antirez/redis, 2019.

[42] "The httperf HTTP load generator," https://github.com/httperf, 2019.

[43] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 1, pp. 29–45, January 2009. [Online]. Available: https://doi.org/10.1109/TSE.2008.103

[44] F. Křikava and 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. [Online]. Available: https://doi.org/10.1145/3213846.3213863

[45] P. D. Marinescu and C. Cadar, "KATCH: High-coverage testing of software patches," in *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, August 2013, pp. 235—-245. [Online]. Available: https://doi.org/10.1145/2491411.2491438

[46] T. Kuchta, H. Palikareva, and C. Cadar, "Shadow Symbolic Execution for testing software patches," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 10:1–10:32, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3208952

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

[48] N. Arora, J. Bell, F. Ivančić, G. Kaiser, and 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. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238186

[49] I. Kravets and 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)*, March 2012. [Online]. Available: http://www.dcs.gla.ac.uk/conferences/resolve12/papers/-session4_paper2.pdf

[50] N. Viennot, S. Nair, and 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. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451130

[51] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2010, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/1811039.1811057

[52] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX Annual Technical Conference (ATC)*, June 2007, pp. 25:1–25:14. [Online]. Available: https://dl.acm.org/doi/10.5555/1364385.1364410

[53] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and 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. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00030

[54] C. Lidbury and 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. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314635

[55] E. Pobee and W. K. Chan, "AggrePlay: Efficient record and replay of multi-threaded 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. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338959

[56] H. Liu, S. Silvestro, W. Wang, C. Tian, and 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. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192380

[57] Y. Shalabi, M. Yan, N. Honarmand, R. B. Lee, and J. Torrellas, "Record-replay architecture as a general security framework," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, February 2018, pp. 180–193. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00025

[58] GDB Wiki, "Process record and replay," 2013. [Online]. Available: https://sourceware.org/gdb/wiki/ProcessRecord

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

[60] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *3rd International Workshop on Dynamic Analysis (WODA)*, 2005, pp. 1—-7. [Online]. Available: https://doi.org/10.1145/1083246.1083251

[61] Y. Hu, T. Azim, and 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. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814320

[62] S. Joshi and A. Orso, "SCARPE: A technique and tool for selective capture and replay of program executions," in *23rd IEEE International Conference on Software Maintenance (ICSM)*, October 2007, pp. 234–243. [Online]. Available: https://doi.org/10.1109/ICSM.2007.4362636

[63] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and 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. [Online]. Available: https://dl.acm.org/doi/10.5555/3291168.3291171

[64] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum, "Automating live update for generic server programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 207–225, March 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2584066

[65] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 612–621. [Online]. Available: https://dl.acm.org/doi/10.5555/2486788.2486869

[66] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and 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. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304054