Ad hoc Test Generation Through Binary Rewriting

Anthony Saieva
Department of Computer Science
Columbia University
New York NY USA
ant@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 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. Many modern record-replay tools 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.

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

I. Introduction

When a security vulnerability or other critical bug is not detected by the developer test suite prior to deployment [1], 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 [2]. However, user validation is also difficult and time-consuming.

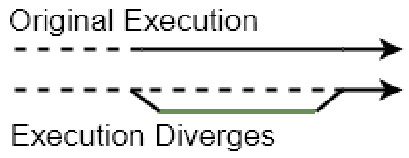


Fig. 1. Ad hoc Test Generation Concept

The existence of the Common Vulnerabilities and Exposures (CVE) list for security vulnerabilities [3] and mundane user bug reports [4], [5] 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 [6], [7], it is common for no such test to be known [8]. 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 as evidence of 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 a test automatically generated from that execution trace. If this test is applied to the same code that produced the original execution trace, the execution will be the same. If the test is applied to modified code, i.e., patched to try to fix the bug, it should execute as if the modified code had been running in the user context instead of the original code. If the patch successfully fixes the bug, then this execution will not manifest the bug.

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 hoc test generation is feasible when execution trace divergence is small, analogous to Tucek et al's "delta execution" [9], 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.

Current record-replay technology replays the recorded execution trace with the original code. Record-replay tools faithfully reproduce not just the externally visible manifestation of the bug but also the intermediate program states for the developer to inspect, single-step, etc. in an interactive debugger or via inserted debugging statements. Current record-replay systems do not support testing prospective patches that modify the program in ways that *change* those intermediate program states.

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 manifested. Unlike existing record-replay tools, ATTUNE leverages binary rewriting [10] 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 leverages two key insights: Our first key insight is that the symbol tables resident in Linux ELF files, intended for linking relocatable code, 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. New functions, etc. can also be accessed by patched functions. 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 changes in patched functions.

ATTUNE follows the workflow illustrated in Figure 2 to generate an ad hoc test for candidate patches that is 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

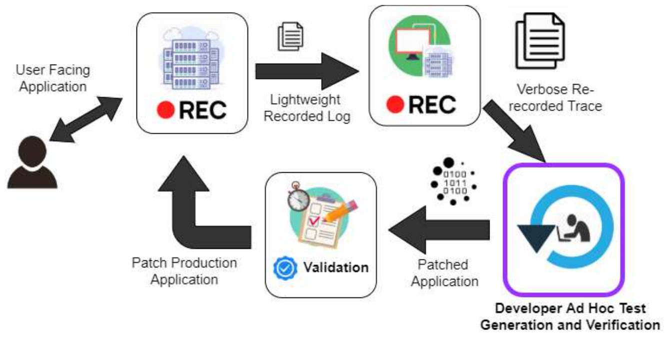

User Facing Application
REC
Lightweight Recorded Log
REC
Verbose Re-recorded Trace
Developer Ad Hoc Test Generation and Verification
Patched Application
Validation
Patch Production Application

Fig. 2. ATTUNE Workflow

Castor [11]. Then when user observation, analysis, monitoring, etc. determines that a lightweight trace manifests a security vulnerability or other bug, then that trace is replayed (offline from production but still in the user environment) and simultaneously re-recorded (analogous to Crosscut [12]), by ATTUNE's verbose recorder.

Our initial ATTUNE prototype builds on the **rr** open-source record-replay tool [13]–[15] from Mozilla as the verbose recorder. (The authors of this paper are not affiliated with the rr developers or Mozilla.) We did not modify rr's recorder and use it as-is to produce detailed execution traces during re-recording. We made substantial modifications to rr's replayer subsystem to generate and run ad hoc tests for prospective patches. rr runs without privileges in user-space on commodity hardware and operating system, assuming commodity compiler, libraries, etc., with no changes to the application's programming language code or executable files. ATTUNE likewise runs without privileges in user-space, with conventional hardware, operating system, compiler, libraries, etc. and no changes to the application. 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.

Unlike third-party website-session script recordings [16], the user or a background user-organization process must initiate re-recording and submit the re-recorded traces to developers; neither ATTUNE nor rr runs surreptitiously. 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 [17]–[19]), and only the detailed but anonymous trace sent to developers, but this is not implemented in the prototype.

Our requirements for verbose execution traces and the technical details of our binary rewriting techniques are explained in Section II. Our evaluation in Section III describes how a developer would use ATTUNE to test candidate patches for a variety of security vulnerabilities and bugs from well-known open-source projects. Section III also gives an example where the user records their own workload with the original program

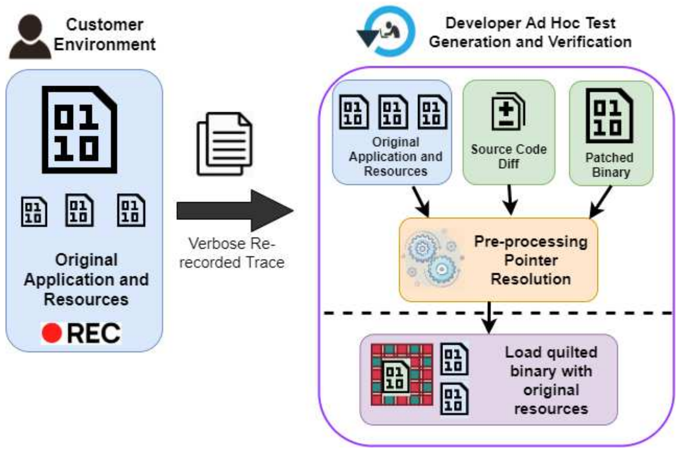

Customer Environment
Original Application and Resources
REC
Verbose Re-recorded Trace
Developer Ad Hoc Test Generation and Verification
Original Application and Resources
Source Code Diff
Patched Binary
Pre-processing Pointer Resolution
Load quilted binary with original resources

Fig. 3. Recording and Preparation for Ad Hoc Test Generation

and replays with the modified program to convince themselves that the bug has been fixed and the patch does not break other behavior. Section IV compares ad hoc test generation to 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. This paper's final version will include a link to our github repository containing ATTUNE's open-source implementation and documentation.

II. Architecture and Design

Our ad hoc test generation workflow constitutes four main procedures: *recording*, *static preprocessing*, *load time quilting* and the *runtime replay decisions*, which we describe in turn. Recording and the two preparation stages are shown in Figure 3, with runtime depicted later on in Figure 11.

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 2. Instead of rr, any other recording engine that constructs sufficiently verbose traces would suffice, but we do not know of any actively-supported open-source alternatives. Specifically, the trace must provide the details needed for ATTUNE to recreate the successive register contents and memory layouts leading up to when the bug manifested. Thus

```
--- a/pngrutil.c // file info
+++ b/pngrutil.c
@@ -3167,10 +3167,13 @@ png_check_chunk_length(...) { //function intfo
  ...
- (png_ptr->width * png_ptr->channels  // source changes
  ...
+    (size_t)png_ptr->width
+    * (size_t)png_ptr->channels
```

Fig. 4. libpng-bug-1 Abbreviated Example Patchfile

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

Fig. 5. libpng-bug-1 Symbol Table Entries

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. An abbreviated example patchfile from a libpng bug fix [20] is shown in Figure 4. 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.

Two mechanisms within the binary allow for 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 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

```
...
<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 // address to line number table
0x0000403a  [3166, 0] NS
0x00004046  [3182, 0] NS
```

Fig. 6. libpng-bug-1 Relevant DWARF Line Entries

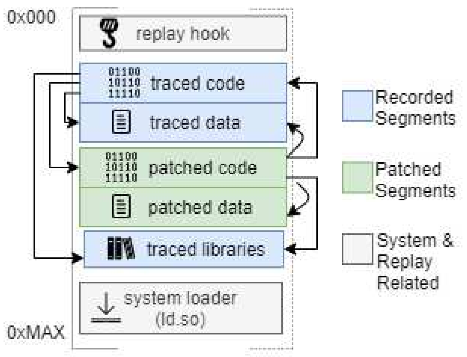


Fig. 7. Address Space Detail

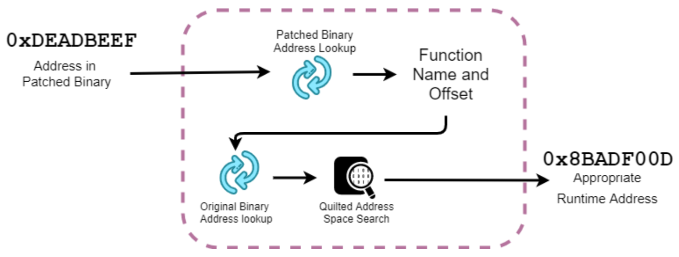

0xDEADBEEF
Address in Patched Binary
Patched Binary Address Lookup
Function Name and Offset
Original Binary Address lookup
Quilted Address Space Search
0x8BADF00D
Appropriate Runtime Address

Fig. 8. Pointer Translation Procedure

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.

C. 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.

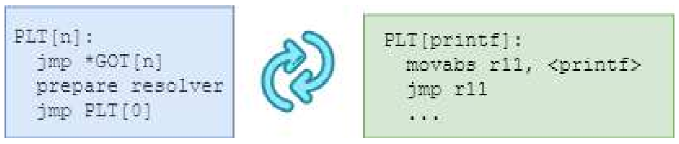
Address Translation Procedure. A summary of the procedure to translate pointers from the context of the modified binary to the context of the original binary is given in Figure 8, and consists of both pre-load and load-time actions. The process starts from the address of the modified function as determined 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

```
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. 9. libpng-bug-1 abbreviated linking example



```
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

```
PLT[printf]:
  movabs r11, <printf>
  jmp r11
  ...
```

Fig. 10. PLT Transformation

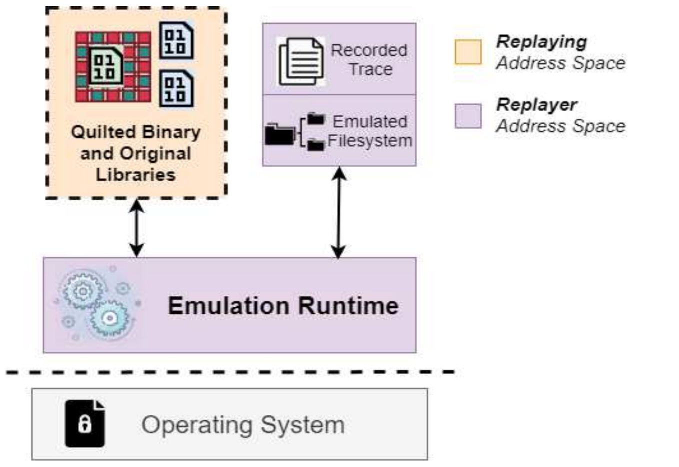


Fig. 11. Runtime Architecture

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 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 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 modifed 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.

D. 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

```
Result getResult(event) {
    if(!diverged) return next_recorded_result;

    if (is_syscall_without_file_io
        && exists unused in log)
            return recorded_result;
    if (is_syscall_with_file_io &
        & supported_recorded_operation)
            return recorded_result;
    if (is_signal && signal_is_recorded) {
        if ( current_pos == inserted code )
            return execute_live();
        else return
            delay_signal_until
               _recorded_RCB_count();
    }

    return execute_live();
}
```

Fig. 12. Runtime Decision Algorithm

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.

E. Limitations

ATTUNE relies on rr as-is to record the execution trace and to replay that recording with the original version of the program [13]–[15]. Since rr was designed to be used during developer testing, with too high overhead for production [13], 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, but Mashtizadeh et al [11] explain this limitation is generally unimportant in practice.

Neither 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. There are other tools available for that purpose [21]–[23]. ATTUNE's role is to test the developer's candidate patches as if they had been in place in the user context where the security vulnerability was discovered.

Our ATTUNE prototype inherits other 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 [13], [14]. 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 bugs.

Independent of rr limitations, ATTUNE also does not support changes to data structures, e.g., changing the size of a struct on the stack or in the heap, that would require changes to memory allocation. 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.

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. This paper's final version will include a link

Bug	Success or Failure	Patching Effort	Files Modified	LOC Changed
curl-1 [24]	✓	Changes how a string is parsed	1	16+, 16-
curl-2 [25]	✓	Changes functions arguments and call.	4	9+, 9-
curl-5 [26]	✓	Modified if statement for buffer overflow	1	4+, 1-
curl-6 [27]	✓	Added new function and inserted call	1	55+, 8-
curl-8 [28]	✗	Changes multiple functions calling a write function	4	17+, 17-
curl-9 [29]	✓	Change parameters to a function call	1	2+,1-
curl-10 [30]	✓	Adds a condition check	1	6+, 2-
curl-11 [31]	✓	Off by 1 correction	1	1+, 1-
curl-12 [32]	✓	Changes libc calls to add extra parsing	1	5+, 5-
libpng-1 [20]	✓	Calculation modification for divide by 0 error	1	6+, 3-
libpng-2 [33]	✓	Adjust calculation for idat chunk max	3	13+, 13-
wc-1 [34]	✓	Added new function and changed condition check	1	23+, 2-
wc-2 [35]	✓	Added error condition check	1	3+, 0-
yes-1 [36]	✓	Substantial changes in option parsing	15	40+, 141-
shred-1 [37]	✗	Removed a break statement	1	1+, 1-
ls-1 [38]	✓	Added condition for change in option parsing	1	1+, 2-
mv-1 [39]	✓	Adding a conditional check before operation	2	6+, 0-
df-1 [40]	✓	Replacing open calls with stat calls	2	12+,8-
bs-1 [41]	✓	Changing a loop condition	1	2+, 1-
wget-1 [42]	✓	Adding conditional check for log	1	1-, 2+
redis-1 [43]	✓	Adding conditional check	1	1+, 1-

Fig. 13. Patch-Testing Dataset

to our github repository containing ATTUNE's open-source implementation and documentation.

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 be patched 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 13.

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 13, resp. We organize the 19 bugs successfully handled into several different types and describe how the developer employs ATTUNE in each case, then explain the 2 failures.

String Parsing bugs are fairly common as there are often many corner cases, which can have significant security implications since input strings may act as attack vectors. Figure 14 [24] 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 uses the recorded test case 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*. Since the log has no information regarding how the network would have responded to the http request had it been sent, the test ends.

Figure 15 [32] deals with mishandling URL strings crafted with special characters, e.g., the "#@" in

```
    ...
+ 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. 14. 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. 15. Curl-12 String Parsing

http://example.com#@evil.com caused Curl to erroneously send a request to a malicious URL. The patch calls *sscanf* with a different filter string. Since the surrounding function handles all the URL parsing for the application, it is rather large with lots of references. Unlike the above bug, which only requires resolving pointers to old strings, the new filter string needs to be loaded into a new data section and referenced appropriately. ATTUNE recreates the state that caused the initial behavior and then jumps to the modified code. There the developer can verify the patch by checking the values in *protobuf* and *slashbuf*.

Mathematical Errors can have security implications when related to pointer errors or integer overflows. For example, an attacker could craft a malicious PNG image that triggers a bad calculation of *row_factor* in Figure 16 [20], 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 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. While not particularly strenuous 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.

```
+/* 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. 17. wc-1 New Function and Refactoring

```
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. 18. redis-1 Erroneous Conditional

A patch for the *wc* file processing utility adds special character parsing functions as shown in Figure 17 [34]. 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 *wc* 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 18 [43] didn't check the client flags for the monitor, which resulted in a kernel panic. While this

```
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. 16. libpng-1 Mathematical Error

```
  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. 19. wget-2 New Loop

was one of the smaller patches, the validation process varied substantially from the log. ATTUNE enables the developer to step through the program and watch progress through the modified control flow past the point of the crash.

New or Changing Loop Conditions. Bad loop conditionals are also common. Reference resolution is performed as before, but these patches vary greatly from an ad hoc testing perspective because loop conditionals do not necessarily exhibit the bug on the loop's first iteration. One such example from the *wget* utility demonstrates how ATTUNE handles this sort of change in a security-critical situation. The bug allowed attackers to inject arbitrary HTTP headers via CRLF sequences into the URL's host subcomponent. Attackers could insert arbitrary cookies and other header info, perhaps granting access to unauthorized resources. The developer modified the *url_parse* functions in Figure 19 [44] 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 [36] and swapped control flow blocks df-1 [40]. 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:** A removed break statement in shred-1 [37] 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 also failed due to **DWARF omissions:** Applying ATTUNE to parameter changing in curl-8 [28] was unsuccessful. ATTUNE failed to locate pieces of the modified function in the loaded binaries and couldn't link a patch. ATTUNE depends on DWARF information for line numbers so test construction was unsuccessful.

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

Quilting Time: Since ATTUNE's quilting occurs at load time, the procedure runs when each candidate patch is tested. The overhead depends on the time it takes to parse the binaries, search for pointers, and update those pointers. If this is excessive, it might hinder developers' ability to produce a successful patch. Figure 20 shows our measurements of quilting overhead. In the worst case the wait time is slightly below 4 seconds and in the best case near instantaneous.

Memory Footprint: Quilting the patched functions into the original binaries adds some overhead to the program's memory footprint during testing. Figure 21 shows specific measurements. The worst case overhead was close to 100KB in curl-2 and a bit over 25KB in curl-11, but otherwise

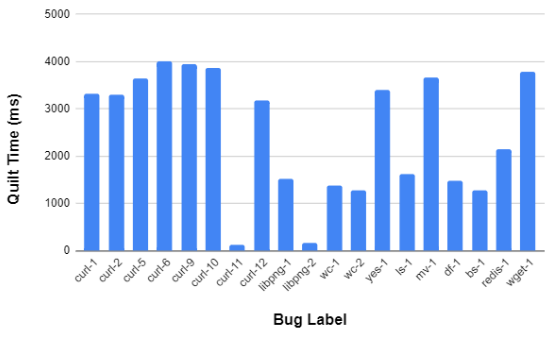


Fig. 20. Quilting Overhead

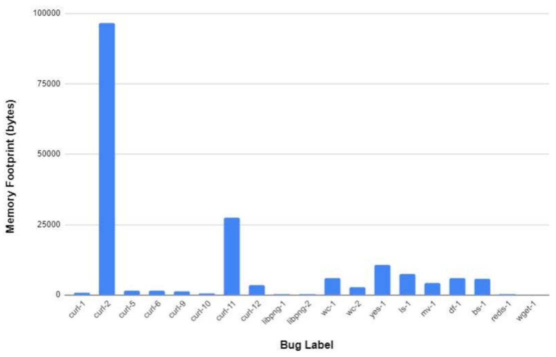


Fig. 21. Quilting Memory Footprint

remained well under 25KB. Modern Linux systems with 64bit address spaces and 4KB page sizes can easily accommodate this overhead. Increases in the program's memory footprint from quilting is not related to the number of lines of code patched (added or deleted), but instead to the size of the functions that have been patched and the associated data those functions access, as well as of course the number of functions patched. For example, the curl-2 patch is so large because it spanned multiple large functions.

C. ATTUNE *enables users to 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. Because ATTUNE operates entirely in user-space, without hardware, operating system, etc. support, it can run in both developer and user environments. ATTUNE leverages the "diffs" in programming language code, ELF and DWARF information during developer testing, which it summarizes and exports into metadata sent along with the released binary patch.

```
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. 22. redis-bug-1 Metadata for User Validation

For sample user environment workloads, we used the redis benchmark [45], which simulates thousands of different requests to the server, and the *httperf* benchmarking tool [46] making thousands of connections. The validation procedure for the redis patch [45] is similar to the redis discussion above, but ATTUNE utilizes only the metadata it added to the released patch, shown in Figure 22.

ATTUNE needs the addresses of inserted and deleted lines for its runtime decision algorithm. The metadata's "inserted line addresses" and "deleted line addresses" are offsets into the relevant files. Deleted lines are removed from the original binary so those addresses 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. With this metadata ATTUNE can emulate the user's original execution trace that demonstrated the bug, to verify it has been fixed, as well as emulate new execution traces from the user's choice of workloads that do not trigger bugs in the original version.

D. 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.

IV. Related Work

Kuchta et al [47] 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 [48] 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., [49], with analogous advantages and disadvantages.

Kravets and Tsafrir [50] proposed "mutable replay", a hypothetical design to construct a new execution trace for a modified program 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 in Dora [51], building on the Scribe record-replay system [52]. Dora leveraged checkpoint/restart [53] 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. The underlying Scribe record-replay 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 [54], but the program itself is not modified. Arora et al [55] 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 [56]–[60]. Some versions of gdb build-in recording and replaying debugging sessions [61], as does Microsoft's IntelliTrace [62]. 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., [63]–[65], 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., [66]–[68]. Cui et al [69] 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 [11] 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" [70]. In MVE, the patched and original versions run simultaneously on production user workloads, adding runtime overhead but enabling immediate detection of undesirable divergences [71], [72]. Unlike MVE running different code versions, as in ATTUNE, LDX [73] runs two instances of the same code to infer causality between execution events. The slave explicitly changes one event from the master execution to find divergent impacts on later events, which is orthogonal to our work.

Fuzzing seeks inputs that induce crashes and other problems [74]. Other approaches also strive to induce bad behaviors, e.g., [75], [76]. [77] builds on EvoSuite's search-based testing [78] to reproduce crashes. Symbolic execution [79] 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 [80]. These kinds of tools, as well as conventional regression testing, are complementary to ATTUNE.

V. Conclusion

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

VI. Acknowledgements

The Programming Systems Laboratory is supported in part by NSF CNS-1563555, CCF-1815494 and CNS-1842456. We would also like to thank Yangruibo Ding, Victor Xu, and Ziao Wang for their work compiling the test data set.

References

[1] Q. Wang, Y. Brun, and A. Orso, “Behavioral Execution Comparison: Are Tests Representative of Field Behavior?” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 321–332.

[2] 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

[3] The MITRE Corporation, “Common Vulnerabilities and Exposures,” 2019. [Online]. Available: https://cve.mitre.org/

[4] 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

[5] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, “Information Needs in Bug Reports: Improving Cooperation Between Developers and Users,” in *ACM Conference on Computer Supported Cooperative Work (CSCW)*, 2010, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/1718918.1718973

[6] 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

[7] hackerone, “Vulnerability disclosure philosophy,” July 2019. [Online]. Available: https://www.hackerone.com/disclosure-guidelines

[8] 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

[9] 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*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508267

[10] 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

[11] 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

[12] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, “Multi-stage Replay with Crosscut,” in *6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’10. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1735997.1736002

[13] 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)*. Santa Clara CA: USENIX Association, Jul. 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[14] R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, “Engineering Record And Replay For Deployability: Extended Technical Report,” *CoRR*, vol. abs/1705.05937, 2017. [Online]. Available: http://arxiv.org/abs/1705.05937

[15] Mozilla, “what rr does.” [Online]. Available: https://rr-project.org/

[16] S. Englehardt, G. Acar, and A. Narayanan, “No boundaries: Exfiltration of personal data by session-replay scripts,” *Freedom to Tinker*, November 15 2017. [Online]. Available: https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/

[17] M. Castro, M. Costa, and J.-P. Martin, “Better Bug Reporting with Better Privacy,” in *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 319–328. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346322

[18] J. Clause and A. Orso, “Camouflage: Automated Anonymization of Field Data,” in *33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985797

[19] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, “Safer Program Behavior Sharing Through Trace Wringing,” in *24th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)s*, 2019, pp. 1059–1072. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304074

[20] Red Hat Bugzilla – Bug 1599943, “libpng-bug-1. 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.

[21] B. Johnson, Y. Brun, and A. Meliou, “Causal Testing: Understanding Defects’ Root Causes,” in *42nd International Conference on Software Engineering (ICSE)*, 2020. [Online]. Available: https://arxiv.org/abs/1809.06991

[22] H. Wang, X. Xie, S.-W. Lin, Y. Lin, Y. Li, S. Qin, Y. Liu, and T. Liu, “Locating Vulnerabilities in Binaries via Memory Layout Recovering,” in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 718—-728. [Online]. Available: https://doi.org/10.1145/3338906.3338966

[23] Y. Kim, S. Hong, and M. Kim, “Target-Driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection,” in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 16—-26. [Online]. Available: https://doi.org/10.1145/3338906.3338934

[24] “curl-bug-1. curl string parsing bug,” https://github.com/curl/curl/pull/3365, 2019.

[25] “curl-bug-2. curl string parsing bug,” https://github.com/curl/curl/pull/3381, 2019.

[26] “curl-bug-5. curl info leak,” https://github.com/curl/curl/pull/3381, 2019, cVE: https://curl.haxx.se/docs/CVE-2017-1000101.html .

[27] “curl-bug-6. curl security vulnerability,” https://github.com/curl/curl/pull/3433, 2019.

[28] “curl-bug-8. curl change parameter fix,” https://github.com/curl/curl/commit/e50a2002, 2019.

[29] “Curl$_follow : acceptnon-supportedschemesfor$”$fake$”$redirects,'' https : //github.com/curl/curl/commit/2c5ec339ea67f43ac370ae77636a0f915cc5f$

[30] “Url: fix ipv6 numeral address parser,” https://github.com/curl/curl/pull/3219, 2018.

[31] “curl-bug-11. curl globbing error,” https://github.com/curl/curl/issues/3251 , 2019.

[32] “curl-bug-12. curl string parsing vulnerability,” https://github.com/curl/curl/commit/3bb273db7 , 2019, cVE: https://curl.haxx.se/docs/CVE-2016-8624.html .

[33] “libpng-bug-2. libpng idat miscalculation,” https://sourceforge.net/p/libpng/bugs/270/ , 2019.

[34] “wc-bug-1. wc special character bug,” https://github.com/coreutils/coreutils/commit/a5202bd5 2019.

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

[36] “yes-bug-1. yes coreutils library function,” https://github.com/coreutils/coreutils/commit/44af84263e, 2019.

[37] “shred-bug-1. shred coreutils library function,” https://github.com/coreutils/coreutils/commit/c34f8d5c787e6, 2019.

[38] “ls -aa shows . and .. in an empty directory;,” https://debbugs.gnu.org/cgi/bugreport.cgi?bug=30963, 2018.

[39] “’cp -n -u’ and ’mv -n -u’ now consistently ignore the -u option,” https://github.com/coreutils/coreutils/commit/7e244891b0c41bbf9f5b5917d1a71c183a8367ac 2018.

[40] “df-bug-1. df coreutils library function,” https://github.com/coreutils/coreutils/commit/b04ce61958c, 2019.

[41] “Running b2sum with –check option, and simply providing a string ”blake2”,” https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28860, 2017.

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

[43] "redis-bug-1. redis monitor request causes crash," https://github.com/antirez/redis/commit/e2c1f80b, 2019.

[44] "wget-bug-2. wget insert new loop to parse url's," https://github.com/mirror/wget/commit/4d729e322fae, 2019, cVE : https://nvd.nist.gov/vuln/detail/CVE-2017-6508.

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

[46] "The httperf http load generator," https://github.com/httperf, 2019.

[47] 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

[48] 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, Jan 2009.

[49] 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

[50] I. Kravets and D. Tsafrir, "Feasibility of Mutable Replay for Automated Regression Testing of Security Updates," in *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)*, London, UK, March 2012. [Online]. Available: http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session4_paper2.pdf

[51] 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

[52] 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*, ser. SIGMETRICS '10. New York, NY, USA: ACM, 2010, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/1811039.1811057

[53] O. Laadan and J. Nieh, "Transparent Checkpoint-restart of Multiple Processes on Commodity Operating Systems," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ser. ATC'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 25:1–25:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364385.1364410

[54] 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: http://dl.acm.org/citation.cfm?id=3291168.3291208

[55] 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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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)*, Feb 2018, pp. 180–193.

[61] GDB Wiki, "Process Record and Replay," 2013. [Online]. Available: https://sourceware.org/gdb/wiki/ProcessRecord

[62] 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

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

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

[65] S. Rattanasuksun, T. Yu, W. Srisa-An, and G. Rothermel, "RRF: A Race Reproduction Framework for Use in Debugging Process-Level Races," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 162–172.

[66] 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

[67] 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

[68] S. Joshi and A. Orso, "SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions," in *23rd IEEE International Conference on Software Maintenance (ICSM)*, Paris, France, October 2007, pp. 234–243.

[69] 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: http://dl.acm.org/citation.cfm?id=3291168.3291171

[70] 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.

[71] 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: http://dl.acm.org/citation.cfm?id=2486788.2486869

[72] 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

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

[74] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-Specific Fuzzing with Waypoints," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 3, October 2019.

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

[76] H. Wu, N. Changhai, J. Petke, Y. Jia, and M. Harman, "An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing," *IEEE Transactions on Software Engineering (TSE)*, 2018.

[77] M. Soltani, A. Panichella, and A. van Deursen, "A Guided Genetic Algorithm for Automated Crash Reproduction," in *39th International Conference on Software Engineering (ICSE)*, 2017, pp. 209–220. [Online]. Available: https://doi.org/10.1109/ICSE.2017.27

[78] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, Apr. 2017. [Online]. Available: https://doi.org/10.1007/s10664-015-9424-2

[79] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Commununications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[80] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A Study on the Lifecycle of Flaky Tests,” in *42nd International Conference on Software Engineering (ICSE)*, May 2020.