# TACHYON: Tandem Execution for Efficient Live Patch Testing

Matthew Maurer
maurer@cmu.edu
*Carnegie Mellon University*

David Brumley
dbrumley@cmu.edu
*Carnegie Mellon University*

## Abstract

The vast number of security incidents are caused by exploits against vulnerabilities for which a patch is already available, but that users simply did not install. Patch installation is often delayed because patches must be tested manually to make sure they do not introduce problems, especially at the enterprise level.

In this paper we propose a new *tandem execution* approach for automated patch testing. Our approach is based on a patch execution consistency model which maintains that a patch is safe to apply if the executions of the pre and post-patch program only differ on attack inputs. Tandem execution runs both pre and post-patch programs simultaneously in order to check for execution consistency. We have implemented our techniques in TACHYON, a system for online patch testing in Linux. TACHYON is able to automatically check and verify patches without source access.

## 1 Introduction

Most attacks target known vulnerabilities for which there are already patches. For example, Microsoft reported that only 0.12activity in the first half of 2011 involved a zero-day attack for which a patch has been available for a month or less [19]. For the other 99.88%, exploits were successful simply because available patches were not installed. These statistics indicate that one of the best ways to reduce security incidents due to exploits is to simply patch vulnerable systems.

The need to rapidly deploy security patches in enterprise environments is hampered by the need to also test patches for problems. Bad patches often have more business risk than a security breach, suggesting that the ability to test patches and guard against such problems might result in faster deployment of security patches. Measures currently deployed in cloud environments deal with random failures, rather than systematic ones caused by bad patches. As a result, current best practices amount to manual testing, which is slow, error-prone, and expensive. For example, NIST best practices recommend manual patch testing (which is slow) on pre-production environments (which are expensive to acquire and maintain) when available, or simply waiting to see if others report a problem or not [18]. While such approaches prevent bad patches from being applied, they increase the vulnerability window. Even when a pre-production environment is provided through virtualization, reducing the cost substantially, auxiliary services, such as databases, must be simulated. This leads to excessive administration overhead and compute overhead. Additionally, effects are captured in an often ad-hoc manner (e.g. by recording network traces), which can miss changes the administrator did not think to look for. For example, the US Air Force implements a centralized patch testing procedure for their half million managed machines, but as a result, delay patch rollout by up to a quarter year [14].

If we could automatically test patches, then we could shorten the vulnerable time window between when a patch is released until it is installed. However, automated patch testing faces several challenges.

First, in order to faithfully check that functionality is preserved in a patch, we should be able to test a patch on the system it will ultimately protect. Second, in order to be widely applicable, we should be able to test patches in the common scenario where the patch is a new binary program, as source is often not available. Third, we want to minimize manual effort. As patches can change the semantics of a program, a human will likely always need to be in the loop to determine if the semantic changes are meaningful. However, we still wish to automate the system as much as possible. Unfortunately, there is very little work on automated patch testing, and no previous work addresses all these requirements.

In this paper we propose the first techniques for live patch testing via tandem execution. Our insight is that current manual testing checks whether the executions of a pre-patch and post-patch binary produce different outputs on known inputs. We call this *observational equivalence* between pre and post-patch.

Tandem execution uses this insight to automate patch testing by simultaneously running both programs on the same input. More specifically, in tandem execution one program runs live on the system (e.g., the patched program), with all system calls (syscalls) being serviced by the kernel. The second version of the program (e.g., the unpatched program) runs in tandem, but with each syscall to the kernel simulated by replaying the side effects from the corresponding calls of the live version. The replay prevents duplicating side-effects, such as writing to the same file twice.

If the two programs deviate on the syscalls issued, or the arguments to syscalls, then they are not observationally equivalent. We record the deviation and inform the user of the potential problem. At this point, the actions that can be taken are to halt the program, or to specify that the deviation is permitted. In order to continue testing, the user provides a rewrite rule that specifies how to handle the deviation, and automated patch testing continues. In our experiments we show that rewrite rules are small when needed, and often completely unnecessary for security-related fixes.

We implement our approach in a system called TACHYON. TACHYON is based upon syscall replay techniques for binary programs, but with a new twist. Existing system call replay schemes are designed to record system calls from one run of a binary for replay against *exactly the same binary*, e.g., [1, 11, 22]. Since both record and replay are against the same binary, the record step only needs to conceptually keep a snapshot of the memory cells affected by the syscall. The affected memory cells are typically determined by differencing the pre and post-syscall memory state. During replay, the memory cells at exactly the same addresses are replayed with the recorded data.

The twist in our setting is we want to replay syscalls to a *different binary*. Typically the two binaries will have a different memory layout, and may make different syscalls. For example, the system may have ASLR enabled, or a patch may change a buffer from the too-small size of 1024 bytes to the just-right size of 4096. In either case, all pointers are likely to have different addresses between the patched and unpatched versions. As a result, previous raw memory snapshot and replay approaches do not work.

To address our twist, TACHYON takes a semantic-based approach to replay only the semantically meaningful information from a syscall in a recording during replay, rather than capturing details like pointer values which may change between patches. Our approach is enabled by three techniques. First, we extend the C type system to include a full description of the syscall side-effects. The description enables TACHYON to identify semantically meaningful arguments and results in syscalls instead of relying on blind memory differencing. The work to annotate the system calls needs to be performed once, and can be reused for all programs. Second, TACHYON utilizes a rewrite rule system to compare syscall sequences for equivalence and rewrite if necessary. The rewrite system gives the end-user the ability to say when deviations are permitted in a systematic manner. Third, TACHYON uses syscall interposition techniques to record the effects of syscalls on a live program, and replay those effects to simulate syscalls on the tandem program.

Tandem execution makes patch testing in new scenarios possible. For example, a test administrator can run the pre-patch binary live and the post-patch in-tandem on the same system. Any deviation reported is either (a) a bug in the patch, or (b) an exploit against the buggy program that is averted in the patched program, or (c) a permitted change that changes IO behavior. In all cases, the administrator should be informed of the deviation. Alternately, a security-conscious administrator may run the patched version live, noting deviations with the pre-patched version. In either case, the tandem execution achieves live patch testing without duplicating side-effects. The closest current best practices come to similar results requires mirroring a production environment with a pre-production environment, which is expensive and requires significant effort to maintain. We discuss other possible applications such as creating honeypots in § 7.

We have implemented and evaluated TACHYON on a number of security patches, and demonstrated that our techniques can successfully detect deviations. We have also performed micro-benchmarks that show our implementation is efficient with respect to the amount of I/O performed. We show our implementation records full syscall information faster than `strace`, a tracing tool targeted at binary programs, and is efficient in comparison to an untraced run.

**Contributions.** The main contribution of this paper is techniques for live tandem execution for patch testing. These techniques automate a large part of patch testing, thus reducing the vulnerability window for unpatched systems. In particular:

- We are the first to propose techniques for automated patch testing that address all the above challenges; we have implemented them in TACHYON. We demonstrate where we run both the unpatched and patched binary and use tandem execution to detect deviations. An additional benefit of tandem execution is that it can utilize extra cores for the security purpose of patch testing.

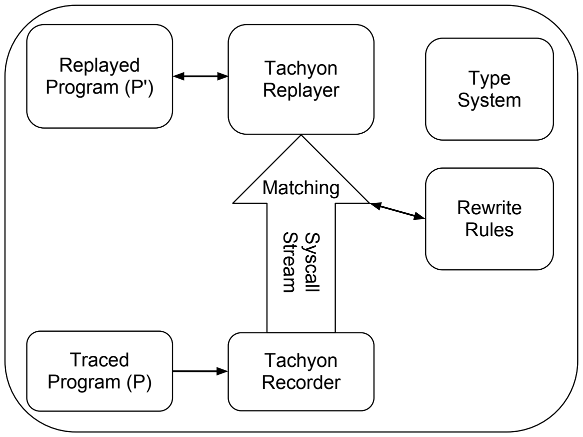

Figure 1: Tachyon System Overview

- We develop a type system that fully encapsulates the side effects of syscalls necessary for replay. Our type system is similar to [11], but does not require source code access or explicit developer cooperation.
- We propose a light-weight rule-based system for checking syscall stream equivalence (and rewriting if necessary) when the sequence of syscalls between the patched and unpatched binary are not exactly the same.
- We have implemented our techniques in TACHYON using Haskell (a type-safe language) and validated the techniques experimentally. Our system is robust enough to handle single-threaded and multi-threaded programs. We evaluate our approach on several real-world patches, as well as synthetic benchmarks, to show the effectiveness and performance of TACHYON.

## 2 Design of TACHYON

### 2.1 Overview of TACHYON and Challenges

The overall architecture of TACHYON is shown in Figure 1. In this paper we call the program running live $P$ (e.g., the patched program) and the program running in-tandem with simulated syscalls $P'$ (e.g., the unpatched program). TACHYON is a user-land program that utilizes the Linux ptrace facility to interpose on syscalls issued by $P$ and $P'$. Like replay schemes, TACHYON has a recorder and a replay module. The recorder records the stream of system calls issued by $P$ and outputs a stream of tuples $\langle C, \vec{I}, \vec{O} \rangle$ where $C$ is the system call number, $\vec{I}$ is a list of inputs to the system call, and $\vec{O}$ is a list of outputs. The replay module interposes on $P'$, and for each syscall $C$ with arguments $\vec{I}$ made by $P'$, simulates the OS by returning $\vec{O}$.

Listing 1: Example patch

```
-int fd = open("/tmp/fileA", O_RDONLY);
+int fd = open("/tmp/fileB", O_RDWR);
-int *storage = malloc(...);
-/* ... Do some processing with storage..  */
+fstat(fd, statBuf);
 char* incoming = malloc(chunksize);
 ssize_t size = read(fd, incoming, chunksize);
 if (size != -1)
   write(fdOut, incoming, size);
```

Consider the example shown in Listing 1, with the patch difference being displayed in diff style with full context. The first edit changes the file opened from /tmp/fileA to /tmp/fileB. The next few edits remove an unneeded call to malloc, and add fstat. The rest of the program is the same. Note that since a malloc call was removed, the returned memory chunk for incoming will be at a different address, even on systems with a deterministic memory layout. Overall, this patch example illustrates three challenges: patches may change arguments to system calls, may change system calls issued, may change memory allocation patterns, and any of these changes may have effects on subsequent execution.

The above challenges motivate three main requirements of live patch testing as distinguished from a normal replay system. First, instead of offline replay, a live patch testing solution should be *online* where $P$ produces the syscall stream that $P'$ should consume. Second, a live patch tester should not depend upon pointers because absolute memory addresses may change between runs. For example, $P'$ and $P$ may issue calls to malloc for different amounts or ASLR may be enabled. Either case prevents patch testing. As a result, we cannot determine $\vec{O}$ by simply diffing the memory state before and after a syscall, as in previous syscall replay schemes [11, 22]. Additionally, memory diffing does not allow us to determine the inputs $\vec{I}$ to system calls. As a good live patch tester should verify the inputs as well, we need some way to extract all inputs of a system call. Without a semantic model, we will be unable to both locate all the relevant components of the input, and to avoid capturing irrelevant components. Thus, we need a semantic model of the inputs. Third, since a patch may remove or add system calls, the live patch testing scheme should allow for the syscall stream to be rewritten during replay. This can be accounted for by allowing rewriting of the tuple stream $\langle C, \vec{I}, \vec{O} \rangle$.

## 2.2 System Calls and Side-Effects

TACHYON needs to determine what the semantic inputs and outputs to a syscall are in order to record and replay them. Specifically, it needs to (1) determine the types of arguments to a syscall, (2) differentiate input from output, and (3) pointers from the pointed-to data. While existing C syscall prototypes are sufficient for (1), they do not provide enough information for (2) and (3). Consider the `read` syscall declaration:

```
ssize_t read(int fd, void *buf, size_t count);
```

This C declaration misses crucial information. First, it gives no clue how the void pointer buf works. How big is it? Is it null-terminated? Are the contents relevant before the call, after, or both? We need to answer all these questions in order to copy the appropriate semantic data. We can see that even the assertion that a pointer points at some data before or after the system call is not the case, as with `sbrk` (pointer points at the end of your address space) and `mmap` (one pointer is only a suggestion). `read` is one of the simple cases; several syscalls have complicated dependencies between input and output parameters, as will be discussed later in §3.

TACHYON addresses the challenges associated with understanding the semantics of syscalls by adding type annotations, as described in §3. The TACHYON annotation language is a light-weight dependent type system that says how to parse the inputs and arguments into semantic data at runtime. These type annotations only need to be written once per system call, and are portable across systems with the same syscall signatures.

## 2.3 Syscall Stream Rewriting

Many patches also change the sequence of syscalls made in addition to the actual parameters. Consider Listing 1. The system call stream when executing the patched program is $\langle ..., \texttt{open}, \texttt{fstat}, \texttt{read}, \texttt{write}, ... \rangle$. However, the call after `open` in the unpatched program is `read`, not `fstat`.

New patched versions often have new system call patterns that cause the program to behave differently at an IO level. It is not possible to tell whether a particular change in system call patterns is valid without a human to validate it. For example, if it turns out the above code is just shifting a few things around and adding a new inconsequential call to `fstat`, then the user may want to ignore the deviation. However, the `fstat` may have been inserted for security, and a deviation may indicate an attack. When opening files in `/tmp/` a common security practice is to then call `fstat` to obtain the user ID and group ID of the file to make sure they are correct in order to detect race conditions. TACHYON should report the deviation and halt execution in such instances.

Although we cannot automatically decide which deviations matter, TACHYON does *automate finding deviations*, as well as provide a mechanism to ignore such deviations when found to continue testing. The rewriting engine relies upon rules that are created for each patched program that detail how to handle semantic differences. For example, if a system administrator decides the above deviation is inconsequential a rule can be written to ignore the `fstat` call. Alternatively, patch creators could write such rules and distribute them with their patches to aid testing.

## 2.4 Road Map

In the rest of this paper, we first describe the TACHYON type system in detail. We then discuss how TACHYON rewrites system calls, as well as some common rules we have found in patches we have tested. We next describe our implementation and evaluation. We finally discuss several applications of TACHYON outside automated patch testing.

## 3 System Call Types

The C function declarations for syscalls do not describe all side-effects. TACHYON proposes an extension to the C type declarations to encode all semantic information necessary to record which parameters are inputs, which are outputs, and how to identify all bytes for each parameter. While this problem has been attacked before [11], our particular needs are different due to the binary only-nature of our approach, as we discuss in § 9.

The TACHYON type system takes advantage of the user-space/kernel-space barrier for interposition. The barrier provides a clean separation that can be monitored without requiring source to the program. In addition, the barrier allows TACHYON to not monitor internal kernel state. The reason is that the only way $P'$ and $P$ can interact with the underlying system is via TACHYON, and TACHYON's mechanism ensures that both programs see an identical state. This is a vital complexity reduction.

TACHYON uses a limited form of lightweight dependent types (types which depend on values). Our lightweight use avoids pitfalls such as undecidability normally associated with dependent types. In the rest of this section, we first provide an intuition on the issues and how dependent types are used, and then describe the full system.

## 3.1 Intuition

A basic approach for recording syscalls is to decorate C types with information about which parameters should be treated as inputs, outputs, or both. We call such annotations the IO class for each parameter. In order to specify how to copy output parameters, we also need to know the size of their values. The size information is needed because we need to copy all output bytes from `buf` in the monitored program $P$ to the address space of $P'$. For example, we could imagine annotating the `read` call as:

```
ssize_t read(int fd, void output* buf{ret},
     size_t count);
```

The parameter `buf` has been annotated as an output parameter, thus should be copied and replayed to $P'$. The annotation also specifies that `ret` bytes should be copied, where `ret` is the return value.

Unfortunately, such simple annotations are insufficient with many data structures, such as vectors. A prime example of a difficult system call is `readv`, which provides vectored reads of a file descriptor. Its C type declaration is:

```
ssize_t readv(int fd, const struct iovec *iov,
     int iovcnt);
struct iovec {
  void  *iov_base; /* Starting address */
  size_t iov_len; /* Num bytes in iov_base */
};
```

The main issue demonstrated is that a complete description of the IO behavior of parameters may refer to other parameters. The `iov_base` length is determined by `iov_len`, and the total number of `iov` items is given by `iovcnt`. `readv` is not alone: it has many friends such as `writev`, `preadv`, and `pwritev`. In order to handle such cases, we need a type system that allows us to express *relationships* between parameter values.

TACHYON uses lightweight dependent types that can express relationships between the value of one parameter and the value of another. We view types as a tree, and use dependent types to walk the tree to determine a value.

The types allow us to walk from the top of the tree, or from the current parameter. In TACHYON, we specify `readv` as:

```
ssize_t readv(int fd, const struct inputoutput
     iovecin *iov{iovcnt}, int iovcnt);
struct iovecin {
  void input *iov_base{undo(self).iov_len};
  size_t iov_len;
}
```

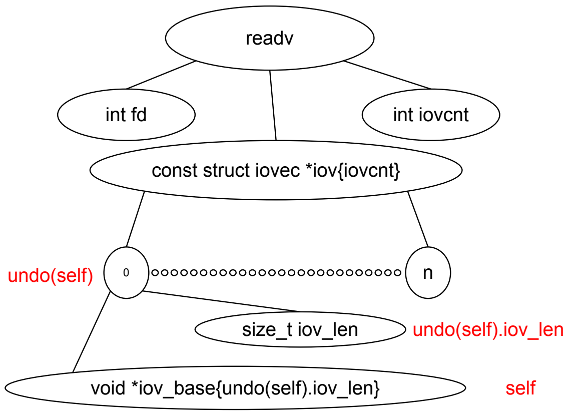

Figure 2: A lookup in action

We now call the struct `iovecin`, because while both `readv` and `writev` take an `iovec`, they are used differently, and so are assigned differing types (specifically, in one case the buffers inside are output, while in the other they are input). The only new annotations compared to before are `undo` and `self`, which are used to walk the type tree to reference other fields. The semantic meaning is that `iov_base` is `iov_len` bytes. `self` refers to the location at which the current value is being read from. `undo` simply says to step back along whatever indexing step was done to get there. In this case, this means that `self` represents the tree traversal up through that instance of `iov_base`. The "undo" brings us up a level, to be looking at the struct. Then, we index the struct to `iov_len` and are done. Figure 2 graphically shows the type tree for `readv` and how the syntax expresses fields in the tree.

## 3.2 The Tachyon Type System

The full TACHYON dependent type system is shown in Figure 3, and is taken directly from the TACHYON source code in Haskell. The language is similar to BNF, where non-terminals are to the left of the equal sign, and brackets denote a list (e.g., [A] is a list with elements of type A).

In TACHYON's language, IOC represents an IO class, that is, whether the pointer is input, output, or both. T represents some form of termination, to allow us to include null-terminated data. NT is for null-terminated data; UT is for unterminated data. If a pointer is null-terminated, reading will cease when a 0 is hit, if this happens prior to the end of the buffer. The index operation is used on both arrays and structs, where the $i$'th index refers to the $i$'th field (counting from 0).

The types available are

- Small - These correspond to basic integer C types,

```
data SysSig = SysSig Type [Type]
data Type = Small Int
          | Struct [Type]
          | Ptr IOC Type Bound T
data T = NT | UT
data IOC = In | Out | InOut
data Bound = Const Int | Lookup Lookup
data Lookup = Ret | Arg Int | Index Int Lookup
    | Self | Undo Lookup
```

Figure 3: The TACHYON annotation language

like `char` or `long`, and indicate values that should not be treated as pointers. The type parameter is the number of bytes of the type, e.g., Small(1) is a 1-byte value corresponding to a `char`.

- Struct - an aggregate of other types. Note that previous replay work treated such types as raw buffers because they could determine size by simply diffing memory before and after a syscall. In live replay, we explicitly lay out all fields because the underlying types may yield further information.
- Ptr - a pointer annotated with an IO class, the type of element it is pointing to, a way to tell how many elements it points to, and whether or not it respects a null termination convention.

We introduce the concept of a "lookup". This is just a series of steps that can be performed from either the arguments of a function in the case of an input or in/out class pointer, or the arguments and return value in the case of an output pointer, to arrive at a memory location or register. This is demonstrated in Figure 2. The Ret and Arg constructs for a lookup are used to allow us to reference the return value or various arguments in a system call, respectively. This is just the generalization of the tree walking described earlier.

Given this, the encoding of a bound as either a constant or a lookup is rather natural. It is the use of this bound that makes us lightly dependently typed—the type depends on the data in question.

Finally, we can build the fundamental structure all this is for—the system signature. A system signature, indicated by SysSig, is what is assigned to each system call in order to allow the tracer to record and play back its effects. The first parameter is the type of the return value, and the second is a list of the types of its arguments.

**Type Checking TACHYON Declarations.** The TACHYON types need only be written once for each system call, and can be reused for any program. However, since they are written manually, we would like to prevent mistakes. In order to achieve this, TACHYON also provides type-checking to make sure the annotations make sense. In particular, TACHYON checks:

1. Bounds are numbers, not pointers or something else.
2. Bounds use only information which is available for the IO class of the pointer (e.g., input class may not use the return value as a size).
3. Output pointers do not contain structure; they are raw data.
4. Types are potentially compatible with the original C type.

These checks ensure annotations which are usable, self-consistent, and match the C type.

## 4 System Call Stream Deviation Detection and Rewriting

Patches often add, delete, or modify new system calls in the original buggy program. Our example in Listing 1 shows all three cases. When the streams of syscalls differ, then the two programs are semantically different. While this means we cannot automatically tell if the differences are meaningful, we can (a) automatically detect deviations and (b) rewrite deviations when informed by the user that the semantic differences are permitted. The heart of detection and rewriting is TACHYON's syscall stream matching and rewriting engine.

### 4.1 Stream Matching

TACHYON uses a rule-based system for rewriting system call streams during execution, designed to be employed by a user of the tracing software to explain to the system what behaviors it should consider equivalent. The rules must consume a sequence of system calls by $P$, and produce a corresponding set of system calls for $P'$ to make in order to allow for writing call results into $P'$ and checking that $P'$ indeed matches the particular equivalence rule.

As we execute, we have two streams of tuples. TACHYON represents the stream from $P$ as $\langle C_i, \vec{I}_i, \vec{O}_i \rangle$, and the stream from $P'$ as $\langle C'_i, \vec{I}'_i, \vec{O}'_i \rangle$. The easy case is when the two programs are semantically equivalent by issuing the same system calls, i.e., $\forall i : C_i = C'_i \wedge \vec{I}_i = \vec{I}'_i$. In this case no rule is needed, and TACHYON will send the corresponding $\vec{O}_i$ for each $\vec{I}_i$ to $P'$.

Any time the syscall input arguments do not line up, TACHYON reports a semantic deviation. In order to permit some deviations, TACHYON provides the ability to rewrite the system call stream. The rewrite engine takes in a set of rewriting rules $f$. Each rewrite rule $f_k$ is a function which takes in $\langle C_i, \vec{I}_i, \vec{O}_i \rangle$ and $\langle C'_i, \vec{I}'_i \rangle$. The rule uses pattern matching to decide if it applies, and if so, returns a pair of equivalent syscall streams to perform a substitution with. After a match, the stream continues to be consumed by the simulated program $P'$.

The overall mechanism can be used for:

- Determining roughly equivalent syscalls, e.g., many small writes being patched to be one big rewrite.
- Ignoring syscalls, e.g., the $P$ program issues a call that is not needed by $P'$.
- Limited reordering, e.g., allowing for syscalls to be switched.

## 4.2 Rewriting Rules

Each rewrite rule $f$ takes a system call (the one made by $P$) and the input to a potential system call made by $P'$, and returns a substitution in the stream. The substitution is implemented as a pair of lists, where the left list indicates the syscalls consumed by the rule, and the right list indicates the corresponding substitution produced by the rule. The type signature for $f$ in TACHYON is:

$$\text{Syscall} \rightarrow \text{SysReq} \rightarrow \text{Maybe ([Syscall], [Syscall])}$$

where the "Maybe" indicates that the rule may also return that no substitution was performed.

The rewriting rules are pure functions, which means they have no access to outside resources like the current syscall stream or application state. By being pure we ensure that rewrite rules can be applied in any order. In addition, it ensures that the rule engine itself will not continually accumulate state, i.e., while individual rewrites may take substantial space, the space used will remain constant in the number of system calls which have gone through, which is vital to an online system.

During execution, the matching engine maintains a queue of syscalls executed by the live program $P$. Suppose the queue contains any syscall $x$ that is not `write`, but the simulated program $P'$ issues a `write` syscall. The simplest rule is to ignore the `write`. This is accomplished by adding a `write` to the queue before $x$. When the matching engine re-examines the queue, it will match the still-pending `write` to the one in the queue, and not report a deviation.

In TACHYON, the rule is written as:

```
1 ignoreWrite :: Syscall
2             -> SysReq
3             -> Maybe ([Syscall], [Syscall])
4 ignoreWrite x (Write 2 buf sz) =
5   Just ([x],
6         [Syscall (Write 2 buf sz) sz, x])
7 ignoreWrite _ _ = Nothing
```

This rule fires when line 4 is matched. This occurs when $P$ issues a syscall $x$ that doesn't match $P'$'s syscall `write`. On line 5 the rule directs it to consume whatever is on the stream at the moment, and replace it on line 6 with the stream of `Write` followed by $x$. This can be thought of as "faking" the call for `Write` to the stream matcher so that it does not report a deviation. On line 7, we catch the case where our conditions are not met, and indicate we did not modify the stream.

The simple, no-look-behind method of replaying with this equivalence is to replay the stream normally until a match fails. At this point, the two syscalls that failed to match are fed into all rewrite rules, and their replacement list for the original stream is checked. If there is still more than one rewrite rule remaining, one is chosen arbitrarily. In future work, checkpointing could be used here to allow for the ability to rewind if the wrong replacement was chosen. In practice, the rules we tested have only yielded one matching rewrite.

A more complex example is what we call write splitting, which occurs when a larger write in the original program is translated into two smaller writes in the replayed program. This is useful if the buffer size used in a transmission was decreased during the patch, as it allows for a roughly equivalent operation—writing one part of the message, then the other—to be treated the same as the original system call writing the entire message. A concrete example would the difference between the program fragments:

```
-#define CHUNK 4096
+#define CHUNK 1024
 while(buf < end) {
   buf += write(fd, buf, min(end - buf,
       CHUNK));
 }
```

In the patched case here, we will see on average 4 times as many system calls, but fundamentally, the same thing is happening. A rewriting rule for write splitting says that a sequence of previous writes can be used to fill on big write request:

```
writeSplit :: Syscall
           -> SysReq
           -> Maybe ([Syscall], [Syscall])
writeSplit (Syscall (Write fd buf sz0) sz)
    --sz is return size
           (Write fd' buf' sz')
 | (fd == fd')
  && (sz' < sz)
  && ((take sz' buf) == buf')
     = Just ([Syscall (Write fd buf sz0) sz],
             [Syscall (Write fd' buf' sz') sz',
              Syscall (Write fd' (drop sz' buf)
                                 (sz0 - sz'))
                      (sz - sz'))
writeSplit _ _ = Nothing
```

This rule states that if we have a write call in the original stream, and the replayed program is trying to make a non-matching write call, but it matches on the file descriptor, and has a smaller size, and the write it is trying to make is a prefix of the original write, then we can replace the original write with two smaller writes, the first of which is the target write, and the second of which is set up to represent the rest of the write.

In line 6, we do a sanity check that the file descriptors we are writing to are the same, followed by a similar check in line 7 that the request from $P'$ has a smaller size than the original form $P$. Finally, in line 8 we make sure that what is trying to be written is a prefix of the appropriate length of the original write. Given these conditions, we know that we can provide a replacement rule which will allow the trace to proceed. In line 9, it tells its caller to consume the most recent call, asserting that it matches the call passed into us. In line 10, we see the first call that is going to end up on the new stream, which matches the input vector we've received from $P'$, and so will allow the trace to continue. In the 11,12, and 13, the function returns an additional item for the system call stream, which represents the rest of the write that has been split. In 14, we catch the case where we don't apply, and simply return no pattern.

## 5 Architecture

TACHYON is built to target Linux for the x86-64 architecture via the `ptrace` call, written in Haskell. An abstraction barrier is in place around `ptrace`, to ensure the technique's generality and portability to other systems with similarly powered tracing libraries. Haskell was chosen for abstract data type support, multi-OS portability, relative speed, and a monadic abstraction layer that proved useful for our tracing environment.

**`ptrace`** TACHYON uses the `ptrace` system call to accomplish system interposition. Use of `ptrace` starts with initialization, in which options are set and the remote process either volunteers itself for tracing, or is traced via a command to attach to its pid. Then, `wait` and `wait4` are used in an event loop to get status information about processes (which are paused when generating one of these messages) and are then resumed at a later time.

The wakeup and sleep powers are implemented by selectively choosing to not resume or resume threads at system call boundaries. While this only enables us to support putting the currently running thread to sleep, we never needed to stop any thread for which we were not currently processing an event.

**Trace Abstraction.** The trace abstraction layer is designed to expose only primitives we believe to be constructible on all platforms for portability purposes. Additionally, the interface was higher level than the tracing interface directly available on most platforms, enabling easier authoring of the code. Fundamentally, a handler is provided for events which the tracing interface detects and sends back. The potential events currently supported are pre/post syscall, and process split. Available to the callback is the ability to put threads to sleep, wake them up, read and write registers, and read or write memory in the target process.

**Multithreading tolerance.** Up until now, we have considered only programs using one thread. However, many modern programs use multiple threads in their normal operation. For example, `curl` in § 6 uses them during DNS lookup.

To deal with threads, TACHYON employs techniques inspired by the field of deterministic multithreading (DMT)[2, 3, 15]. A DMT mechanism is one that makes a program insensitive to the scheduler as an input. That is, given the same inputs other than kernel scheduler action, it will yield the same outputs.

In TACHYON, we enforce an ordering over system call events. We differentiate from a mismatched system call in need of rewriting and a thread being early or late by choosing to block the thread if the thread IDs on the syscalls don't match, and invoke the rewrite engine if they do match, but the system calls don't. This forms a looser notion of consistency than is used in regular DMT, but is sufficient for our purposes. However, applying a real DMT system in addition to our techniques would likely yield an even more robust treatment of threading able to deal with shared memory data transfers and other such intricacies, as we discuss in § 8.

**Special Syscalls.** While in general the simulated application $P'$ uses the effects of syscalls from the live $P$, TACHYON does have a few exceptions. The exceptions occur when a syscall result from $P$ cannot be emulated in $P'$. This usually occurs when there is something which is part of the thread life cycle or virtual memory system, which are not facilities that can be directly accessed by TACHYON. Luckily, there are only a limited number of cases.

The first is `sbrk()`, which is the syscall responsible for dynamic memory allocation. Luckily, this particular call can be passed through, as it does not modify OS state, it only serves to modify the process's VM system.

When a `clone()` occurs, we match input arguments like any other system call, and then allow it through. `ptrace` feeds us an event notifying us as soon as the new thread exists, and pauses that thread before it can do anything so that we can attach to it. This event is also part of the synchronized system call stream. TACHYON then registers that new thread and its pair between emulated tid and real tid, and proceeds with normal operation.

The `exit_group()` system call works like any other, except that it acts as an end-of trace marker. We currently only accept full application exit, in which all threads are simultaneously terminated, but there is no reason our techniques could not be extended to individual thread destruction.

The most complicated is `mmap()`, which maps a file or device into memory. Our implementation depends upon the operations performed on the region. The read-/write case would be extremely expensive to monitor, as all writes are effects, so we would have to interpose on every memory write to that page, and so this case is disallowed entirely. It might be possible to deal with writable mapped files or shared regions shared outside the program, using page faults and slow execution, or some form of snapshot trick. We leave this as future work.

In the read only case however, the translation is straightforward. Rather than issuing the mapping as requested, we instead simply ask for a private mapping of the same size this thread received during recording, and fill that buffer with the contents that memory contained after the initial map. Private mappings of anonymous memory are also easy to support, and are be simply passed through. Finally, in the case of shared memory, we can allow it if it is both anonymous and our multi-threading system is in place. This could be extended to allow for non-anonymous shared memory by spawning fake file descriptors to the region, but is left as future work.

Other system calls for which we would need to add special implementation to allow them to be serviced by the kernel, but still safely matched include `munmap`, `mprotect`, `fork`, `sigaction`, `sigreturn`, and `exit`. These were unneeded for our test cases and were not implemented.

## 6 Evaluation

We have evaluated TACHYON with respect to three main questions. First, can TACHYON detect deviations where the patched program is semantically different than the unpatched, and how hard is it to write rules to ignore deviations that do not matter? Second, what is the performance factors for TACHYON, including best and worst case settings? Third, what is the performance on real programs? In this section, we describe our results.

| Program | Issue ID | Description |
|---|---|---|
| cURL | CVE-2011-2192 | Improper key delegation |
| mplayer | EDB-ID 11792 | Table index out of bounds |
| php5 | CVE-2012-0832 | Bad Argument handling |
| php5 | CVE-2011-1938 | Buffer Overflow |
| ncompress | CVE-2001-1413 | Buffer overflow |
| htget | CVE-2004-0852 | Buffer overflow |
| gs | CVE-2010-1869 | Buffer Overflow |
| glftpd | EDB-ID-476 | Buffer Overflow |
| socat | CVE-2004-1484 | Format String |
| corehttp | CVE-2009-3586 | Off-by-one Buffer |

Figure 4: Successfully Detected Deviations for Security Patches

### 6.1 Detecting Deviations

To test the effectiveness of TACHYON, we used it on real patches to detect known deviations. The patches we tested are shown in Figure 4. In this experiment, we tested the program on normal inputs, and verified that TACHYON did not report a deviation. We then tested on inputs that triggered known deviations, e.g., exploits in the original program or bugs in the patch.

For cURL, the patched vulnerability was an information disclosure bug. In the unpatched version, Kerberos credentials were (accidentally) forwarded instead of just a proof the user was authorized. We verified that the unpatched program would send credentials, and the patched program did not. In order to test the patch and allow normal operation of safe inputs, we had to write two rules for cURL that totaled 11 lines. The rules were necessary because cURL added a non-security feature that affected file descriptors in their patch.

CVE-2011-4885 addresses a problem in PHP where hash collisions are easy to find, which can be used to launch a remote denial of service attack. TACHYON required no rewrite rules to run the patch on safe inputs. The patch, however, broke the argument handling for arrays after loading many arguments. CVE-2012-083 addressed this problem. For CVE-2012-083, we again required no rewrite rules for safe inputs.

CVE-2011-4885 and CVE-2012-0832 demonstrate a patch that is broken, and provide motivation for TACHYON. Since CVE-2011-4885 fixed a purported vulnerability, it should be applied immediately. However, after applying CVE-2011-4885, a *new* vulnerability is introduced. TACHYON detects those new exploits as deviations immediately. In particular, we checked exploits (addressed in CVE-2012-0832) that were unknown in

2011, and verified that they caused detected deviations. Thus, if an administrator had been running TACHYON, and immediately applied the patch, they would detect exploits immediately against the vulnerability introduced.

The EDB-ID 11792, CVE-2001-1412, and CVE-2004-0852 all patch typical security bugs by adding inline checks. These checks did not change the system call pattern or arguments, thus no rules were needed for patch testing.

For CVE-2010-1869, `gs`'s memory problems required a rewrite rule to admit additional or skipped calls to `brk`. 8 lines were required for these rewrite rules. Three lines were required for EDB-ID-476 to allow for rewriting of the format of a usage message. Four were required to deal with the new `lstats` in the patch for CVE-2009-3586.

## 6.2 False Positive Testing

To show that TACHYON is fairly precise, we tested it on the most recent 207 patches to coreutils. (The number 207 was chosen because that was how far backwards we could go easily with an automated building system.) From this, we found that in 18 cases out of 1656 executions, a deviation was reported, or TACHYON crashed. Looking at the output, 16 of these were TACHYON bugs, but are not systematic, so that re-running the test produced correct results. 2 of these were actual deviations. In the first, a call to `fadvise()` was introduced into `cp`. An equivalence can be reached with a one-line rewrite rule. In the second, a buffer size was changed. The read/write splitting/coalescing rules described earlier in this paper allow an equivalence to be reached. Overall, this indicates that while TACHYON is not perfectly bug free, it never reported a deviation when one had not happened, and deviations that should be acceptable could be easily expressed in the rule system.

We also ran TACHYON on patches for two common utilities with no known vulnerabilities: `/bin/ls` and `/bin/cat`, and used them interactively. In the one month testing period, TACHYON was able to use tandem execution on these utilities for normal day-to-day use with no perceived slowdown. Further, TACHYON reported no deviations (i.e., had no false positives).

## 6.3 Micro-benchmarks

TACHYON has three main sources of overhead: our approach to syscall interposition, transferring bytes from the source to the sync application, and running both $P'$ and $P$ in tandem. Overall we measured a linear overhead for both data transfer and system calls, and 0% of CPU time, as detailed below.

**Syscall interposition overhead.** TACHYON is a user-space system call interposition scheme, which imposes additional context switches but provides for a clean separation of interposition, kernel, and user-space application. Our user-space interposition has 4 context switches per call issued by the live application $P$. For each syscall in $P$, TACHYON context switches from $P$ to the kernel, from the kernel to TACHYON, from TACHYON back to the kernel, and finally back to $P$. Normal operation only has two context switches: from user to kernel space, and back again.

In order to test the effect of these two extra switches, we wrote a simple program that executed `getpid()` in a tight loop. We were sure to call the system call directly, as the standard version of `getpid()` in C actually caches its result.

**Data copy overhead.** TACHYON needs to copy the output data from $P$ to $P'$. A first naive implementation actually incurred 6 copies. TACHYON originally copied output data from $P$ into TACHYON for syscall rewriting, and then copied it to $P'$. Each copy between systems was actually two copies: one from the user-space into kernel space, and one from kernel-space into user-space. This lead to a huge slowdown in early benchmarks (over 6x). To reduce this, we added the ability to the Linux kernel to map a remote process's memory via `/proc/pid/mem` under most situations, making the normal case of reading from the remote process only have one copy.

In Figure 5, we see the overhead is in a linear relationship. The overhead here is the total overhead time for the system. While they make a difference for small data transfers, they are rapidly dominated. This can be seen by the rapid transition to a tight grouping around a linear relationship.

In Figure 6, we again observe a nice linear relationship, showing no residual effects on performance from processing a system call. It shows that it takes less than a second of overhead to process 60,000 syscalls.

When varying the CPU load of the traced program, no noticeable difference in execution time was noticed, as we do not intercept regular computation, only system calls.

Given this, if it is known how many system calls are used, how much data is being transferred, and how much time is being spent on the CPU, we can model how long a given workload would take under our tracer. TACHYON previously incurred a large number of copies and control transfers to move buffers around in comparison with the register fetching it does for simple system calls, and the remote memory fetch path is not optimized in the OS. However, a kernel patch allowing for `mmap` to be used on the special file `/proc/pid/mem` considerably ameliorate this, resulting in the new statistics above.

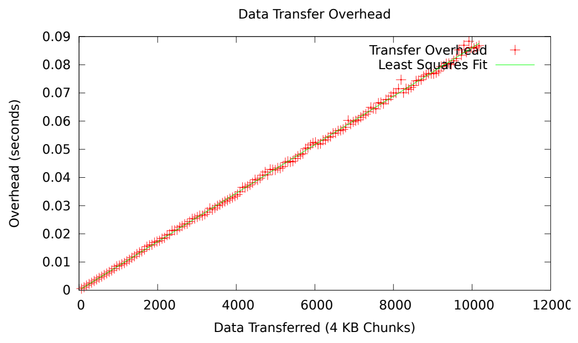

Figure 5: Data Transfer Overhead

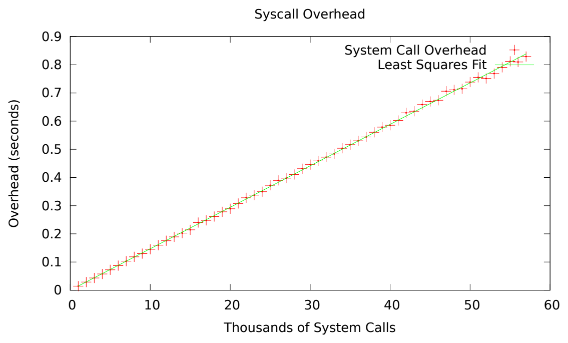

Figure 6: Syscall Overhead

**Tandem execution CPU time.** The final source of overhead is in CPU operations. Since TACHYON sleeps when system calls are not being issued, it does not slow down applications that are CPU-intensive. This is a significant advantage over instruction-level interposition tools such as Pin [16] and Valgrind [21], which typically suffer at least several times overhead.

However, we are running both $P'$ and $P$. We verified that TACHYON can utilize independent cores to run both programs with no additional overhead (other than the memory transfer and syscall overhead measured above). Thus, we conclude that TACHYON can utilize multi-core to test patches.

**Efficiency on real programs.** To test the efficiency of TACHYON interposition, we measured its tandem execution against `strace` and `gdb`'s reverse execution. `strace` is a tool built on top of `ptrace` used to monitor system calls. `gdb` allows programs to back-step through operations.[1] Note these tools have different goals than TACHYON; we only use them to evaluate performance.

`gdb`'s replay mechanism derived from its reversible debugging support. However, it proved wholly unsuitable for regions of more than a few instructions. Due to a lack of SSE support, memcpy would be improperly rewound and replayed. Additionally, even then the recording overhead was more than 100x native execution, and built up a huge memory data structure, making it impractical to benchmark.

The results compared to `strace` are shown in Figure 7. Overall, TACHYON was faster, sometimes by a large margin, than a comparable syscall interposition scheme. This is partially because TACHYON can and does process some of its overhead while the traced program is doing work, but mostly due to the massively

| Program | Load | TACHYON | `strace` |
|---|---|---|---|
| `compress` | 32M Random | 1.41 | 19.78 |
| `primegaps` | First 35 | 1.00 | 1.00 |
| `mencoder` | h264 | 1.07 | 1.12 |

Figure 7: Tracing Performance (relative to native execution)

improved facility for retrieving remote memory via our patch to mmap /proc/pid/mem.

**Web Server Tests.** We also tested the throughput of `lighttpd` and `thttpd` when monitored under TACHYON. In this test, we use ApacheBench (ab) configured to make 1000 requests in two threads, downloading 4096 byte web page. We ran the experiment in two scenarios: on localhost and across the Internet.

In the network experiment, the web servers ran at one university, and requests were made from another university on the opposite US coast. There was no detectable degradation for `thttpd`, and only about a 30% slowdown for `lighttpd`. Essentially, what this shows is that while the system does not deal well with applications whose progress is primarily based on the rate at which they can issue system calls, when we move closer to a real deployment, applications do not tend to have that as their primary limiting factor.

In the second experiment, we ran ab on localhost. This is a worst-case test because both web servers spin in a tight loop on a syscall (`lighttpd` spins on `epoll`, and `thttpd` on `poll`). This creates a pathological case for TACHYON, because the application spends most of its time neither doing IO, nor doing computation, but instead spends most of its time moving across the system call barrier.

TACHYON took 8.9 times longer on `lighttpd`

[1]We were unable to test against what is likely the most similar system, R2 [11], as it is both Windows only and requires build-time support (as well as not being public).

(throughput decreased to 10% of original values) and 14.2 times longer on `thttpd` (throughput decreased to 7% of original values).

To round things out, we also ran a test over a few hops on the local network. As expected, intermediate results were measured to be between the two, with 1.17 times untraced completion to complete the test with `thttpd`, and 3.72 times untraced completion to complete the test with `lighttpd`.

With a more real network (or a more complicated webapp), we can see the slowdown is lessened. With a real network, `epoll` will spend more time waiting, diminishing the perceived effects.

## 7 Discussion

**Other Patch Testing Scenarios.** While throughout this paper we have focused on online patch testing where the patched version is run live, we could also run the unpatched version live. We note that the live program can continue executing after a deviation, but currently the syscall sync application cannot. Thus, by running the patched live, we are assuming that after a deviation the right thing is to continue executing the patched version. However, by running the unpatched version live, we can check for incompatibilities while allowing for the original program to continue executing after a deviation.

**Honeypots.** TACHYON can also be used as a type of lightweight honeypot. Let $P$ be a patch for a security vulnerability, and $P'$ be the vulnerable program. Observe that $P$ and $P'$ differ on exploits by definition. By running $P$ and $P'$ in-tandem, TACHYON will report a deviation on attacks.

A clever approach to running a honeypot is to run $P$ as the live program, with $P'$ as the sync. In this setting an attacker only seeing the buggy program. TACHYON will report attacks, e.g., by logging a deviation when shellcode tries to execute `/bin/sh`. However, the system is safe from a real compromise because TACHYON can be configured to abort execution after the deviation.

**Debugging.** One of the most difficult to debug classes of bugs is commonly known as heisenbugs. These are bugs which will seemingly randomly occur or not occur with all of the inputs the programmer knows about held constant. These traces, and the associated replay mechanism, provide a way to step through the program in a completely deterministic way, so that once a heisenbug has been caught with tracing on, it has been captured and the sequence leading to it can be carefully explored and debugged. As we capture all inputs, this also makes it possible for the programmer to debug a crash that took place on another machine, without having to try to replicate the OS state to reproduce the crash.

**Efficiency.** Recall TACHYON uses user-land syscall interposition, and has our approach to syscall interposition as its primary source of overhead Currently, interposing on each system call on the live program requires 4 context switches. TACHYON context switches from $P$ to the kernel, from the kernel to TACHYON, from TACHYON back to the kernel, and finally back to $P$. Normal operation only has two context switches: from user to kernel space, and back again. A kernel-space interposition scheme would also have only two switches.

Recall from § 6 the overhead from copying data is almost linear in the amount data transferred between $P$ and $P'$. A basic in-kernel approach would still have a linear overhead (since data has to be copied into both virtual memory spaces), but likely with a smaller constant factor.

Our user-land approach was chosen because it offers a clean separation of functionality, isn't kernel dependent, and offers an easier development environment. Moving the system call interposition into the kernel would not have these advantages, but would likely improve performance. We leave further study of in-kernel tandem execution schemes as future work.

## 8 Limitations and Future Work

TACHYON could be extended to provide better determinism for shared memory. At the moment, because TACHYON does not schedule individual memory operations, multithreaded programs which have concurrency bugs could run differently under TACHYON. (Non-concurrency bugs are not a problem.) One approach would be to incorporate recent advances in DMT, e.g., [2, 3, 8, 15] into TACHYON. This would also allow for effectful shared memory.

TACHYON currently does not support virtual dynamically-linked shared objects (vDSO), a section of kernel memory mapped into the user-space process to allow for more efficient calls. Unfortunately, some system calls made through a vDSO do not trigger the `ptrace` trap. However, vDSOs are known to provide increased efficiency, so being able to trap that interface could be an improvement, and limit host system modification.

## 9 Related Work

Our approach is motivated by existing replay systems. At a high level, previous work in this area has focused on system call replay (e.g., [11, 22]), virtual-machine level replay (e.g., [12, 17, 24]), and instruction-level replay schemes (e.g., [1, 10, 22]). These systems address the related but different problem of replay against the same binary, e.g., for debugging, while we want to replay to a different binary for patch testing.

Delta Execution [23] uses a similar insight to us for testing, namely that patches tend to change very little, and the majority of the program should remain the same. To accomplish this, they structure execution so that it splits every place in which the patch modified the code, and attempts to merge the execution afterwards, checking that the overall state change during the split matched appropriately. Their approach has the additional advantage of avoiding duplicate computation. TACHYON differentiates itself from this work primarily in its generality; specifically, it works at a binary-only level, it allows matching global effects (e.g. heap changes and IO), can be configured to allow specific non-matching global effects, and allows for structure size changing. Fundamentally, Delta Execution attacks the problem at the level of matching internal state, while TACHYON attacks it from the point of view of observational equivalence from the outside world.

Where Delta Execution places their consistency level inside the application state, which is more specific than us, Capo [20] attacks it from the point of what signals are coming into and going out of the computer. This wins them several benefits, namely the ability to deal with fewer effects and a lesser need for a rewrite or matching system (for example, coalescing or splitting reads or writes is free). Unfortunately, this system also needs specialized hardware and in-kernel code to operate. While we have used kernel code to accelerate TACHYON under Linux, it is not required or fundamental to the technique.

Like TACHYON, R2 [11] designed a type system to express all side-effects, but for the purpose of describing intercepted APIs at the source level. R2 [11] differs from TACHYON in that it targets developers, interposes at the function level (thus requires source), and replays recorded syscalls against the same binary. Although the problem setting is different, there are numerous good ideas that could be borrowed for live replay if source was available. For example, R2 proposed analyzing the call graph to find efficient cut points at which to perform interposition, while we always interpose at every syscall. Unfortunately, we at the binary only level cannot easily prove that a set of interposition points form a complete cut. Additionally, R2's annotation language is actually too powerful, and allows for the expression of types that we cannot appropriately interact with at a binary-only level without compile-time assistance, as it allowed for the computation of arbitrary expressions. We instead limited ourself to navigating trees of pointers, which turns out to be sufficient for the vast majority of system calls.

We record system calls and discover divergences when system call requests do not line up. Another approach would be to perform replay at the instruction level, which would be useful for pinpointing the first point of divergence. This could be done by augmenting instruction-level replay systems like PinPlay [22], and gdb[10] 7.0 and above to take into account differences in memory layout. In undodb [1], memory snapshots and individually optimized system calls are used to accomplish reverse execution.

Alternatively, one could utilize VM playback mechanisms [9, 12, 24] to simulate patches at the whole-machine level. However, testing then requires accurately replaying very low level events. We chose system calls over instruction level or whole-machine level because system call interposition is significantly cheaper, thus more amenable to end-user deployment scenarios. Additionally, rewriting system calls is much more reasonable than rewriting low level events.

Another recent idea in interacting with multiple programs which should meet the same specification, similar to our patched and unpatched pair, is the idea of N-Variant systems [7]. It intends to increase reliability by forcing any exploit or otherwise bad input sent as input to cause the same bad effect in other versions of the software in order to actually occur. There are some similarities here, but our techniques are aimed at fundamentally differing process images, while theirs are aimed at the same underlying code put together in different ways.

Our system does not find new inputs for patch testing. While we assume live or recorded inputs, one could replay both systems on automatically generated inputs as well. For example, we could use test cases produced by automated systems such as KLEE [6], BitBlaze [4], and BAP [13].

Brumley et al. have previously proposed deviation detection at the binary level [5]. The main goal in this work is to automatically *generate* likely deviations, which is a different problem than tandem execution. Once a candidate deviation is generated, the deviation was manually validated (Section 3.3 [5]). Our approach could be used to validate deviations automatically at the syscall level.

## 10 Conclusion

In this paper, we presented TACHYON, a system for testing binary-only patches on real inputs in a live system. We have demonstrated an efficient way to describe interface boundaries on C-style declarations using a lightweight dependent type system. Our experiments show TACHYON is able to automatically detect deviations on real programs. We also suggest our techniques may apply to other problem domains, such as honeypots.

## References

[1] undodb. http://undo-software.com, Nov. 2011.

[2] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–16.

[3] Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News 38* (March 2010), 53–64.

[4] BitBlaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[5] Brumley, D., Caballero, J., Liang, Z., Newsome, J., and Song, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Symposium* (Boston, MA, Aug. 2007).

[6] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation* (2008).

[7] Cox, B., Evans, D., Filipi, A., Rowanhill, J., and Hu, W. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium* (2006), no. August, pp. 1–16.

[8] Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 337–351.

[9] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI* (2002), pp. 211–224.

[10] FSF. *Documentation for GDB*, 7.3.1 ed., Sept. 2011.

[11] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008), pp. 193–208.

[12] Herrod, S. The amazing vm record/replay feature in vmware workstation 6. http://communities.vmware.com/community/vmtn/cto/steve/blog/2007/04/18/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6, Apr. 2007.

[13] Jager, I., Avgerinos, T., Schwartz, E., and Brumley, D. BAP: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification* (2011).

[14] Lane, B. A. The USAF standard desktop configuration (SDC). download.microsoft.com, June 2007.

[15] Liu, T., Curtsinger, C., and Berger, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 327–336.

[16] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (June 2005).

[17] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35* (February 2002), 50–58.

[18] Mell, P., Bergeron, T., and Henning, D. *Creating a Patch and Vulnerability Management Program*. National Institution of Standards and Technology, Nov. 2005.

[19] Microsoft. Microsoft security intelligence report vol. 11. Tech. rep., Microsoft, 2011.

[20] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo. *ACM SIGPLAN Notices 44*, 3 (Feb. 2009), 73.

[21] Nethercote, N., and Seward, J. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification* (Boulder, Colorado, USA, July 2003).

[22] Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 2–11.

[23] Tucek, J., Xiong, W., and Zhou, Y. Efficient online validation with delta execution. *ACM SIGPLAN Notices 44*, 3 (Feb. 2009), 193.

[24] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., and Inc, V. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS* (2007).