Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata

Mathias Payer
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Time Of Check To Time Of Use (TOCTTOU) race conditions for file accesses in user-space applications are a common problem in Unix-like systems. The mapping between filename and inode and device is volatile and can provide the necessary preconditions for an exploit. Applications use filenames as the primary attribute to identify files but the mapping between filenames and inode and device can be changed by an attacker.

DynaRace is an approach that protects *unmodified* applications from file-based TOCTTOU race conditions. DynaRace uses a transparent mapping cache that keeps additional state and metadata for each accessed file in the application. The combination of file state and the current system call type are used to decide if (i) the metadata is updated or (ii) the correctness of the metadata is enforced between consecutive system calls.

DynaRace uses user-mode path resolution internally to resolve individual file atoms. Each file atom is verified or updated according to the associated state in the mapping cache. More specifically, DynaRace protects against race conditions for all file-based system calls, by replacing the unsafe system calls with a set of safe system calls that utilize the mapping cache. The system call is executed only if the state transition is allowed and the information in the mapping cache matches.

DynaRace deterministically solves the problem of file-based race conditions for unmodified applications and removes an attacker's ability to exploit the TOCTTOU race condition. DynaRace detects injected alternate inode and device pairs and terminates the application.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; D.3.4 [*Programming Languages*]: Processors — Run-time environments

General Terms Security

Keywords Security, TOCTTOU races, Race protection, File-based TOCTTOU race protection, Dynamic protection, Virtualization

VEE'12, March 3–4, 2012, London, England, UK.

1. Introduction

Attacks to local security by privilege escalation are a challenging problem because a potential attacker already has access to some privileges on the same machine. An attacker can use these privileges to exploit local bugs in an application to escalate privileges and/or to impersonate a different user.

File-based race conditions are a typical form of Time Of Check To Time Of Use (TOCTTOU) attacks [4, 5]. The application process leaves a window of opportunity to the attacker where a specific bug is exploitable. A second process may interfere on a shared resource whereas the original process assumes that it is using the resource exclusively. These interferences can be used to replace accessed files between checks or, e.g., to deadlock privileged processes. As multicore systems become more and more popular, we expect an increase in attacks that employ multiple (concurrently) executing processes. Multicore systems enable an attack to run alongside the original program; the attacking process can then use cache effects, or other timing issues to exploit a process.

In a Unix-like system file accesses are particularly prone to race conditions. Potential race conditions arise because the mapping from a filename to a unique inode and device number can change. Filenames are volatile and the corresponding inode and device number can change upon every system call invocation. If a path and filename are passed to a system call then the kernel dynamically resolves the path and filename. The kernel then maps the resolved inode and device number (which corresponds to the file at that moment in time when the system call was executed) to a file descriptor that is then passed back to the application. The mapping from inode and device number to a file descriptor is unique and race-free but the mapping from filename to inode and device number is volatile.

File-based race conditions can occur either in the path to the file (i.e., a directory is a symbolic link that can be changed by the attacker), or in the final file atom (i.e., the last file atom is an attacker-controlled symbolic link). The POSIX community tried to address the problem of file-based races by introducing new system calls that work relative to a specific file descriptor [1, 2]. These new system calls can be used to, e.g., test permissions of files relative to a given base directory. A programmer can use these system calls to, e.g., authorize a directory (e.g., using user-mode path resolution) and then safely create a file atom in that directory provided that the directory is validated appropriately. If used properly the new system calls solve the problem of race conditions in the path to the final file atom, but they do not solve the problem of race conditions if the final file atom is a symbolic link.

Much work has been done by the system security community to resolve TOCTTOU races [3, 9–13, 15, 16, 18–20, 22, 23, 27–33, 35–37]. Recently a practical, portable solution has been found that can deterministically solve the problem without requiring mod-

ifications to the kernel or its API: user-mode path resolution [30]. This solution, however, suffers from a serious drawback: a programmer must manually identify and modify all pairs of system calls that are vulnerable to TOCTTOU races.

DynaRace builds on and extends user-mode path resolution by removing the burden of manual program modification. DynaRace deterministically performs safe user-mode file path resolution[1] for *unmodified* applications by applying some user-defined action to every atom along the file path in a race-free manner. The key idea is that DynaRace defines user-defined actions such that they guarantee that the metadata of file atoms do not change between invocations of vulnerable system call sequences. DynaRace then dynamically intercepts and replaces these vulnerable sequences with their deterministically-safe alternative. To this end, DynaRace maintains a cache holding the metadata of accessed files and a state machine that quickly identifies periods of vulnerability.

DynaRace is an approach that protects unmodified applications from file-based TOCTTOU race conditions. DynaRace adds a hidden layer between the unmodified application and the operating system that keeps state and metadata in a mapping cache for all accessed files. This mapping cache is used to verify the integrity and consistency of consecutive file accesses according to a state machine. Each accessed file has an associated state. The mapping cache is either updated with the current metadata or the existing metadata is enforced for the accessed file according to the state of the file. Each file can be in four different states: *new* for new files, *update* for files where the metadata is updated between system calls, *enforce* for files where the cached metadata is enforced between system calls, and *retire* for files that are no longer used in the application.

Individual unsafe system calls like `access` followed by `open` are redirected dynamically to race-free implementations. These race-free implementations are part of the DynaRace framework. They check the state of the file according to the internal mapping cache and the state machine. If the system call executes without a race condition then the mapping cache is updated and the result of the system call is returned to the application.

Our implementation prototype uses user-space binary translation and system call interposition to weave a deterministic race protection framework into the application. An important aspect of our implementation prototype is that neither the application nor any library must be aware of the safe system calls. The system call replacement system works in the background of the process without any coordination. The user-space application can use unsafe system calls and our approach ensures that no file system race conditions are possible.

The contributions of this paper are as follows:

1. DynaRace, a lightweight approach to protect unmodified applications from file-based race conditions.
2. An evaluation and discussion of a DynaRace prototype implementation.

DynaRace keeps state and metadata for each accessed file in an intermediate layer between the application and the operating system. DynaRace redirects all file-based system calls to a stateful inspection framework. This inspection framework uses a state machine and the cached metadata to validate that the metadata of a file has not changed between system calls that test file properties and system calls that use or change files or file metadata.

[1] See [30] for a detailed description of user-mode path resolution.

2. Attack model and background information

This section describes the attack model used for the DynaRace approach and presents background information on file-based TOCTTOU race conditions. File-based race conditions enable an attacker to escalate privileges, e.g., to access, read, or write privileged files.

2.1 Attack model

An attacker has user-access to the system and tries to escalate privileges. The user has no root-privileges and the attacker has no direct hardware access. The DynaRace protection runs with the same privileges as the protected application. A successful attack breaches the security of the application and enables additional privileges to the attacker. DynaRace terminates the application if an attack is detected; denial of service attacks are not part of the attack model.

2.2 File-based TOCTTOU race conditions

Table 1 shows a classic TOCTTOU race condition presented by Mazières et al. in [19]. A privileged (SUID) garbage collector script cleans the `/tmp` directory. An attacker prepares a temporary directory containing an empty file with the same name like the target file. The script reads the directory and relies on the fact that the directory will not change between individual system calls. The attacker removes the temporary directory and replaces it with a link to an existing directory. The script then deletes the file with the same name in the privileged directory.

SUID program	Attacker
	`mkdir("/tmp/x")`
	`creat("/tmp/x/passwd")`
`readdir("/tmp")`	
`lstat("/tmp/x")`	
`readdir("/tmp/x")`	
	`rename("/tmp/x", "/tmp/y")`
	`symlink("/etc", "/tmp/x")`
`unlink("/tmp/x/passwd")`	

Table 1. A potential attacker can race against the privileged application and delete a critical system file.

Table 2 shows a typical TOCTTOU race condition presented in [5]. The `access` system call is used to check permissions of the user in a SUID program. A privileged application relies on the fact that the mapping from a file to an inode and device id remains the same over subsequent system calls. An attacker replaces the file with a link to a privileged file. The privileged application then uses the sensitive file instead of the original file that was authenticated.

SUID program	Attacker
`access("file")`	
	`unlink("file")`
	`link("sensitive", "file")`
`fd = open("file")`	
`read(fd, ...)`	

Table 2. A potential attacker can race against the privileged application and read a privileged file.

3. The DynaRace approach

DynaRace adds a hidden abstraction layer between the application and the operating system. This abstraction layer keeps track of all file-based system calls and keeps information about accessed files and directories in a metadata cache. The monitoring of all file-based

system calls allows DynaRace to keep track of all directories and files that are accessed. Our implementation prototype extends a dynamic system call interposition framework to dynamically monitor all file-related system calls of an application as well as all loaded libraries.

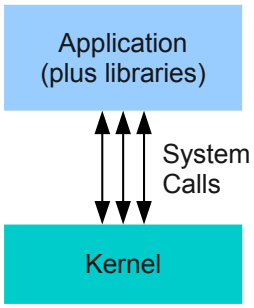

Figure 1. Execution *without* DynaRace checks for file-related system calls.

Figure 1 shows a system without DynaRace. Potentially unsafe system calls are executed directly by the kernel without additional checks for race conditions. Each system call is executed in isolation and without knowledge of the results of prior system calls. Applications without special checks (e.g., storing file state internally, or checking for safe accesses using hardness amplification, see Section 7.5) for these unsafe system calls are prone to race conditions.

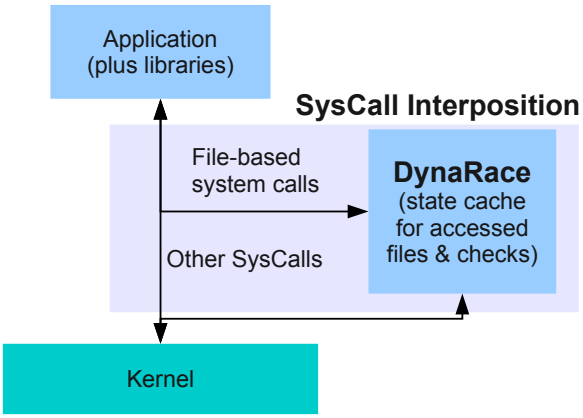

Figure 2. Overview of the DynaRace approach.

Figure 2 shows the DynaRace system. All file-related system calls are intercepted using system call interposition and are redirected to the DynaRace module. DynaRace keeps state for each accessed file and dynamically checks for file-based race conditions. Unsafe system calls like `access` followed by `open` are replaced dynamically with the new and safe race-free sequence of system calls. File-based system calls are no longer executed in isolation but use the state and metadata from prior system calls to validate the current system call. Neither the application nor any library must be aware of the safe system calls. The system call replacement system works in the background of the process without any coordination. The user-space application can use unsafe system calls and DynaRace ensures that no file system race conditions are possible. Due to the replacement strategy of system calls DynaRace can be applied to pre-existing libraries and applications. The DynaRace module works as an independent additional module and can be used in combination with other approaches.

Filenames and directory names can be changed by a concurrent process, e.g., by using symbolic links to redirect a file to a different location. The presented approach uses two security concepts: (i) A chain of trust of known directories from the root directory (*/*) to the directory of the file is constructed by iterating through all the different file atoms. This chain enables the use of the new system calls that are relative to a specific directory. (ii) A thread-local transparent mapping of file statistics that keeps track of all accessed, opened, stated, and modified files.

3.1 File states

The general rewriting policy for unsafe system calls replaces an unsafe system call with a set of dynamic atomic checks. These checks ensure that if a filename is equal to an already accessed filename then the following `open` system call can rely on the subsystem to ensure that the files are also equal.

Equality is defined as *same inode number, device id, and parent directory* for existing files. For files that do not exist equality is defined as *same parent directory and an error code*. The file metadata that is stored in the mapping cache must contain the complete metadata to show equality between different files or, if the file does not exist, information about the base directory and the fact that the file does not exist. The metadata of a file includes a pointer to the metadata of the directory of the file.

Every file that is accessed has an associated state. Depending on the system call that is executed on the file and the state of the file different checks are executed.

Figure 3 shows the different states and the checks that are executed during a transition. Nodes correspond to the states with associated checks and edges to different groups of system calls that force transitions between states. The three states are *update*, *enforce*, and *retire*. A file can be in any of these three states if the same filename has already been used in the program, or in the start state *new* if the file is used the first time.

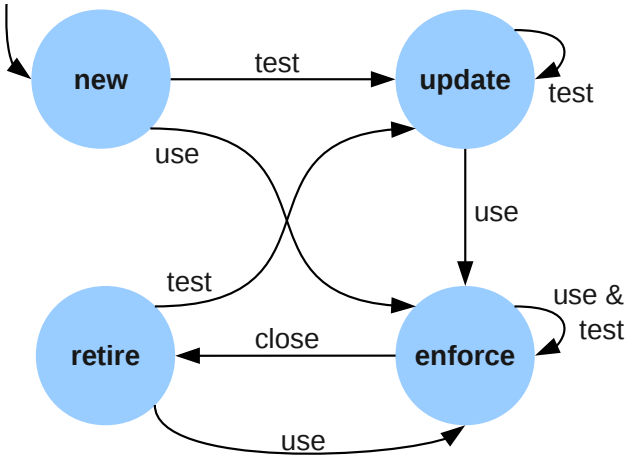

Figure 3. State machine with file states and dynamic checks.

The edges of the state machine correspond to different groups of system calls. The transition from one state to another depends on the system call and the current state. The system calls are separated into different sets depending on their function:

Test: the application uses system calls in this set to gather information and to check file metadata (e.g., permissions, or status).

The system calls in this group do not change the file or any associated metadata. Examples for *test* system calls are `access` or any of the `stat` system calls.

Use: this group of system calls works with files or changes file-related metadata. The system calls in this group modify the file or the associated metadata. Examples for *use* system calls are `open`, `creat`, or `chmod`.

Close: the group of system calls that closes or deletes files: `close`, `unlink`, and `unlinkat`. The application uses these system calls to close or delete files. These system calls signal the kernel that the application no longer works with the specified files.

Depending on the prior state of the file and the new state (as determined by the system call) a set of checks is executed. The metadata is updated if the file has not been changed by the application (e.g., in the states new, update, and retire). If the file transitions from the update state to the enforce state (or stays in the enforce state), then DynaRace enforces the correctness of the available metadata. If the file is a new file that has not been used by the application then DynaRace adds the metadata into the metadata cache. If the file has been closed then DynaRace updates the metadata and warns if there is a mismatch in the metadata cache. This "check and warn" case is used for files that have already been used in the application but were closed.

The state machine in Figure 3 is updated depending on the type of the system call.

Update: all state transitions that end in the update state (*new* $\rightarrow$ *update*, *retire* $\rightarrow$ *update*, and *update* $\rightarrow$ *update*) update the information of a file in the metadata cache or create a new entry for a new file in the metadata cache. This state gathers information about the different files used in the application.

Enforce: whenever a file is used (i.e., either the file itself or the metadata of the file changes) in the application then the state of the file changes to enforce. Enforce ensures that the application can always work with the same file. DynaRace enforces equality if valid information about the file is available in the metadata cache (this holds for the transitions *update* $\rightarrow$ *enforce* and *enforce* $\rightarrow$ *enforce*). If a system call of the use group is used for a new file (*new* $\rightarrow$ *enforce*) then DynaRace adds the information to the cache and changes the state of the file to enforce. If a file has been closed by the application and a system call of the use group is executed (*retire* $\rightarrow$ *enforce*) then DynaRace checks and updates the metadata and emits a warning if there is a mismatch between the cached and the current information.

Retire: a file ends in this state if it has been closed or discarded by the application (*enforce* $\rightarrow$ *retire*). This special state discards and retires files that are no longer used by the application after they have been in the enforce state.

The retire state offers the possibility to retire files that are no longer used by the application. A retired file can be changed by a concurrent process without a security violation. This feature is important to enable ownerships transfers like, e.g., log rotation. Process P_1 checks and opens a log file for writing. A second process P_2 rotates the logs (compresses the open log and creates a new, empty log file). Process P_2 then signals process P_1 to reopen its log files. Process P_1 closes the log file (thereby retiring the metadata information), checks, and reopens the new file.

3.2 File resolution

All file-based system calls are replaced by safe handler functions that dynamically use the mapping cache to verify that identical filenames conform to the same file identified by a unique inode number and device id. File resolution is performed in a manner similar to the `chk_use` algorithm presented by Tsafrir et al. [29, 30]. DynaRace uses the user-defined function in the `chk_use` algorithm to verify and update the mapping cache as the file is resolved atom by atom. The handler functions resolve files in the following way:

1. Split path names into (i) a directory path that contains the directory and (ii) the actual filename.
2. Rewrite relative directory path names to absolute paths according to the current working directory.
3. Identify and check the directory using the directory path.
   (a) If the directory path is already in the mapping cache then the handler function can open the directory and verify that the current directory and the data in the mapping cache are identical.
   (b) Otherwise the handler function builds a chain of trust from the root of the file system to the directory path by opening every single directory on the way and adding the information about the directory to the mapping cache, e.g., for `/var/tmp/file` the directories `/`, `var` relative to `/`, and `tmp` relative to `/var` are checked and added to the mapping cache.

   This function returns a file descriptor that can be used for system calls relative to the verified directory.
4. Next the filename is resolved using the resolved and checked directory. The file is opened and parameters are checked in the handler function (using the new relative system calls `openat` and `fstat64`). If the file is already in the mapping cache then the handler function ensures that the current file and the cached metadata is identical. Otherwise the new file information is added to the mapping cache.

If there is a mismatch between any metadata of either a directory or the file then the application is terminated with a race warning. After the file is resolved the system call handler is executed. This handler might then add new file information to the mapping cache or change existing information, according to the system call that is executed.

4. Implementation

The current prototype implementation of DynaRace relies on several software layers. A binary instrumentation toolkit detects and rewrites all unsafe system calls related to file handling. These system calls are then redirected transparently to the DynaRace module. The application (and all libraries used by the application) still issue the original (unsafe) system calls. Using binary rewriting, DynaRace keeps track of all accessed directories and files.

DynaRace extends the interposition layer from the libdetox framework [21] to check all system calls that modify files. All system calls that execute unsafe file operations are redirected to special handler functions. Each handler function handles one system call. These handler functions either (i) update the mapping cache between filenames and inode and device information for files that have not been used before, or (ii) use available information in the mapping cache to ensure that the filename still maps to the same physical file.

Related work [15, 28, 35] often matches specific predefined pairs of system calls to identify potential races. DynaRace introduces a new mapping cache that keeps track of all accessed files; all file-related system calls are rewritten to use this new mapping cache. This approach detects potential race conditions between any combination of file-related system calls.

The handler functions first resolve the file according to Section 3.2 and then check the actual system call or rewrite specific parameters. Handler functions exist for the following system calls:

Test: the following system calls are in the *test* group:

- **stat*:** all `stat` related system calls (e.g., `stat`, `stat64`) are rewritten to ensure that the specified file ends up in the cache. `fstat64` is used as the actual system call.
- **access:** this unsafe system call has no replacement that uses a file descriptor, so the handler function implements this system call using `fstat64`.

Use: the following system calls are in the *use* group:

- **open:** check flags, if `O_CREAT` is used then the handler ensures that `O_EXCL` is set as well, handling potential errors. `openat` is used as the actual system call, relative to the current directory.
- **creat:** reuses the check for the open system call.
- **chmod:** the handler ensures that the current metadata of the modified file is equal to the cached metadata and changes the system call to use `fchmod` with the tested file.

Close: the following system call is in the *close* group:

- **close:** the handler closes the file and reduces the number of open instances of the current file. If the counter reaches 0 then the file enters the *retire* state.

Section 3.1 explains the different states in more detail. The current implementation prototype emits a warning and terminates the application if an unimplemented system call is used, but the implementation can easily be extended to include other system calls as well.

DynaRace is released as open source. The prototype implementation is included as a module of the libdetox binary translation framework. The source code can be downloaded from the libdetox homepage at `http://nebelwelt.net/projects/libdetox`.

4.1 Tracking of file states and metadata

A transparent file mapping cache keeps the information of all used files and directories. This file mapping cache enables a secure way to identify files and is used in the handler functions of unsafe system calls. These handler functions rely on the file mapping cache to identify files that have already been used in earlier system calls, e.g., a rewritten `open` system call that is preceded by a (rewritten) `access` system call uses the file mapping cache to ensure that if the filenames passed to the system calls are equal then the files themselves are also equal.

The data structure for file entries contains the following fields:

- **state:** the current state of the file. Possible states for files are either *update*, *enforce*, *retire*, or *new* (see Figure 3). Directories are in one of two possible states. A directory is either accessible or in an error state.
- **nropen:** the number of open file descriptors for this file that are in use by the application. A file can only transition to the *retire* state if all open instances of a file are closed.
- **fd:** this field contains either 0 if the file is OK, the open file descriptor if the file is currently opened for DynaRace checks, or the error code if the file is invalid.
- **filename:** a string that contains the file atom for files or the full path for directories.
- **stat:** holds the result of the `fstat64` system call when the file was last accessed. This field is used to check equality of files.
- **dir:** a pointer to a file data structure that contains information about the directory of the file. A file is always verified alongside the directory that it is in.

The file mapping cache is constructed lazily. Whenever a new file is used by the application then the file's metadata and state are added to the mapping cache. If a file or directory is reused then the current state of the file system must conform to the data in the mapping cache. The handler functions update the mapping cache for system calls that change metadata after the system call is executed but before control is returned to the translated application.

4.2 File resolution

File resolution is split into two steps according to Section 3.2. The first step constructs an authenticated chain from the directory of the file to the root of the file system to authenticate the path. The second step uses the base directory to authenticate the remaining file atom (the last component in the path) against the already authenticated base directory using the new file-based race-free system calls. The following sections discuss the implementations to resolve directories and file atoms.

4.2.1 Directory authentication

Path authentication starts with a full path and recursively authenticates single directories moving towards the root directory. The recursive function first checks if the current directory is already in the cache, otherwise a new cache entry is constructed. Newly constructed entries are linked to the parent directory, and the correct parent directory is also verified. Each directory is stored in the cache with the full path to enable a fast lookup.

Directory authentication is implemented using an approach similar to `chk_use` [30]. `chk_use` consumes a path one atom at a time, starting with the root directory. The intention of `chk_use` is to specify a check function (e.g., `access`), and a use function (e.g., `open`) which are then executed after another in a race-free way. DynaRace uses the `chk_use` function to authenticate the directory by traversing the path one atom at a time and inserting the metadata information of each sub-path into the cache.

A notable difference to the original `chk_use` implementation is that DynaRace uses the new system calls like `openat` to traverse the directory path. This setup removes the need to change the current working directory of the process in the `chk_use` function and allows DynaRace to support multiple concurrent threads.

The path authentication function checks each directory once. If the check fails then an error is recorded in the list of paths. If there is an error during the authentication of the chain then this error is propagated upwards to the initial directory (and to the caller as well).

Path authentication takes a full absolute path as an argument and returns either a valid open directory or an error. The opened directory can be used for further authentication of files in that directory.

4.2.2 File authentication

File authentication takes an authenticated directory and authenticates a file atom in that directory. This function keeps track of the state of individual files and validates correctness according to Figure 3. New files are initialized and added to the list of accessed files, existing files are authenticated according to their current state and the target state.

File authentication first searches the list of already accessed files using the authenticated directory and the file name. If there is a cache hit then the file metadata is either updated or verified according to the state of the file. If the directory and the file are not

in the cache (i.e., this combination is used for the first time) then a new entry is constructed with the available information.

This function throws two types of warnings. The first type warns if a file was changed by an external process during the runtime of this process. This type of warnings shows potential attacks against the program that were fixed. The second type warns if files are used without validation (e.g., opening files without checking the permissions first). This type shows protocol violations by the application. If a race attack is detected (i.e., the cached information changes in the *enforce* state) then the application is terminated.

4.3 Replacing system calls

The system call interposition framework checks all system calls and redirects all file-based system calls to handler functions. These handler functions implement the DynaRace core and keep state for each accessed file and all used directories.

The handler functions use directory authentication from Section 4.2.1 and file authentication from Section 4.2.2 to update the state cache of individual files and directories. These authentication functions abstract the bookkeeping problem and enable clean and simple handler functions.

The current implementation provides handler functions for the `stat`, `access`, `open`, `creat`, `chmod`, and `close` system calls. This section shows one system call from each state, `access` for the *update* state, `open` for the *enforce* state, and `close` for the *retire* state. Functions of the form `systemcall_int` (e.g., `fstatat_int`, `openat_int`, `close_int`) are helper functions in the virtualization layer that execute the real system calls.

The prototype implements a subset of all file-based system calls. Missing system calls can be added using the available authentication functions and the existing handler functions.

4.3.1 access system call

The `access` system call handler intercepts `access` system calls and rewrites them dynamically to use `fstat64`. The handler function takes the given filename and splits it into an absolute pathname (resolved using `getcwd` if the path is relative) and a file atom.

The absolute path is authenticated according to Section 4.2.1. All directories on the path are consequently added to the directory cache. Directory authentication returns an open file descriptor for the directory of the used file. This file descriptor is then used to execute a `fstat64` system call. The stat information is needed to construct the state information of the file atom. The file metadata entry is then constructed depending on the return values of directory authentication and the following `stat` system call using the file authentication function.

The `stat64` struct contains all information needed to reimplement the `access` system call. A macro uses the stat information, the user id, and the group id to return the same information like the access system call. The return value of the `access` handler function is either an error value for invalid files or 0. The state update can trigger warning messages that are logged.

4.3.2 open system call

The `open` system call handler intercepts all `open` system calls and rewrites them to use the safe alternatives. The given pathname is split into a file atom and an absolute path similar to the `access` system call handler. The absolute path to the file atom is then authenticated using directory authentication.

The file atom is then opened using the `openat` system call relative to the authenticated base directory. If the application uses the truncate attribute to remove the file's contents then the attribute is removed from the executed `open` system call. The truncating is delayed until after the authentication and the update of the metadata. The handler function then updates file metadata using the file authentication function.

The handler function either returns an error value or the open file handle. If there are authentication errors then the program terminates. If the application opens unchecked files that are in the "new" state then DynaRace emits a warning (in the file authentication function).

4.3.3 close system call

Similar to the `access` and the `open` system calls, the `close` system call is redirected to a handler function. The `close` system call has the advantage that the file is already open and there are no potential race conditions when accessing this open file descriptor.

The handler function searches the cache using the unique inode number and device id. If the file is opened multiple times then the number of open files is reduced. If the last file descriptor of a specific file is closed then the state of the file is updated to the special *retire* state.

4.3.4 New system calls

DynaRace can coexist with applications that already use new system calls that use relative directory file descriptors (`accessat`, `openat`, `fstatat64`, etc.). The new system calls are handled just like regular file-based system calls and redirected to a handler function by the interposition framework. The handler function then updates the file metadata and executes the new system call using the information provided by the application.

5. Implementation alternatives

Our prototype implementation of DynaRace uses and extends the libdetox binary rewriting framework for the execution of untrusted code. Three other implementation approaches are possible. The first approach extends the Linux kernel and implements the DynaRace approach, e.g., as a kernel module. The second approach extends the standard libc and implements the DynaRace approach on top of the library. The third approach uses the `ptrace` debugging framework to implement DynaRace in a concurrent process.

5.1 Kernel DynaRace implementation

An alternative implementation could extend the kernel with a module that implements the DynaRace approach and keeps a cache of accessed files on a per-application basis. This state cache could then be used whenever the application requests an unsafe system call.

An advantage of this implementation approach is that there is no overhead for binary translation and that the kernel can also keep track of all accessed files for all running applications.

A disadvantage is that a kernel-based implementation needs kernel level access. The kernel-based implementation can contain bugs and lead to exploits. Another point is that the Linux kernel already provides file-based system calls relative to open directories. These system calls can be used to implement safe applications. DynaRace is only needed for potentially unsafe applications. New functionality should only be added to the kernel if a user-space implementation is not feasible.

We argue that the risk of potentially exploitable code in the kernel is not worth the advantage of the lower overhead. In addition a user-space implementation is preferable if safe alternatives already exist in the kernel like the set of safe system calls that DynaRace uses.

5.2 libc-based DynaRace implementation

A second alternative implementation could extend the standard libc library. This library contains wrappers for most of the file-based

system calls. These wrappers are then used by the application. The implementation would extend the wrappers for all file-based system calls to implement the DynaRace approach. A static cache of accessed files and states would be initialized during the standard libc initialization and used whenever a file is accessed.

An advantage is the lower overhead compared to our prototype implementation because no binary translation is needed to redirect and catch the system calls.

A disadvantage is that the application can still execute native system calls in using an unpatched ("third-party") library. This third-party library breaks the security of the race detection. A second potential problem is that the DynaRace race checks are executed at the same privilege level as the application. Our implementation uses binary translation for the application; the file-based system calls are then intercepted from the sandbox and redirected to the DynaRace implementation in the privileged part of the sandbox in user-space.

We argue that the risk of a third-party library that executes a direct system call is too high compared to the performance overhead for binary translation. A possible implementation should offer "complete" protection for all system calls, and not only the system calls that are executed through the libc.

5.3 ptrace-based DynaRace implementation

A third alternative implementation leverages the `ptrace`[2] system call to implement the DynaRace approach. A separate DynaRace process controls the target process and observes all system calls externally. The DynaRace process intercepts and inspects all file-related system calls.

DynaRace replaces the unsafe system call with a piece of code that executes a set of safe system calls and some checks. The DynaRace process would inject that code into the running process and redirect the execution flow to the injected code. This injected code would then replace the original system call and return the result from the kernel to the unmodified application code.

An advantage of this approach is that the application does not need to be translated and virtualized. Depending on the implementation of the DynaRace process this setup could lead to some performance speed-up compared to our prototype implementation.

A disadvantage is that the DynaRace process must either stop the traced process upon every system call, a setup that leads to high context-switching overhead, or DynaRace must inject new code into the application that could lead to unwanted side-effects. The implementation of the DynaRace process and the code-injection technique is critical to keep the overhead low. Original system calls must be permanently redirected to injected code to reduce the task switching overhead between the DynaRace process and the application process. Another disadvantage is that the application process does not profit from the additional protection that the sandbox offers.

We argue that a feasible `ptrace`-based implementation will be too complex due to the need to inject code in the application domain. A virtualization-based implementation adds new code naturally and instruments the executed system calls using simple redirection. Binary rewriting tools provide additional opportunities for error checking that are hard to replicate with external debugging techniques like `ptrace`.

[2] The `ptrace` system call is used to remotely control a process. The tracing process can read and write memory locations and registers of the target process, controls signals and signal delivery, and can inspect system calls and parameters.

6. Evaluation

The DynaRace prototype is implemented on top of the user-space security toolkit libdetox. This section evaluates (i) the raw performance of the DynaRace prototype implementation using several microbenchmarks, (ii) end-to-end performance of the DynaRace prototype implementation using a complex Apache setup, and (iii) evaluates DynaRace using several application scenarios. The discussion of the application scenarios shows how DynaRace protects from file-based race attacks in these scenarios.

6.1 Performance evaluation

The set of features and the security guarantees raise the question about the total overhead for libdetox. Optimizations in the dynamic translation process result in an average overhead of between 6% to 9.3% for the SPEC CPU2006 benchmarks, depending on the set of security guarantees that is used [21].

The prototype implementation of DynaRace is not yet optimized and there is potential to reduce the number of executed system calls and to use better data-structures for the mapping cache. Table 3 shows overheads for specific microbenchmarks depicted in Listing 1. Every microbenchmark executes the code sequence 1,000,000 times in a loop. The microbenchmarks are executed on an Intel Core i7 950 CPU at 3.07GHz using a 64bit version of Ubuntu 10.10 and the most recent libdetox version 0.3.0. The benchmarks are compiled using gcc version 4.5.2.

```
/* 1) long test */
if (access("input", R_OK|W_OK)==0) {
  int fw = creat("test", 660);
  if (fw == -1) perror("Creat");
    int fr = open("input", 0);
    if (fr == -1) perror("Open");
    close(fw);
    close(fr);
} else {
  printf("Unable to open dir");
}

/* 2) access test */
int j = access("input", R_OK|W_OK);

/* 3) open/close test */
int j = open("input");
close(j);
```

Listing 1. Microbenchmarks used for the evaluation (C code).

	Native	libdetox	DynaRace
1) long test	4.73ms	4.00ms	20.96ms (4.4x)
2) access test	1.59ms	1.62ms	6.11ms (3.8x)
3) open/close test	1.93ms	1.00ms	6.90ms (3.6x)

Table 3. Results of the microbenchmarks compared to native performance.

An interesting result of Table 3 is that libdetox is able to outperform the native execution for raw system call throughput in the open/close test. Two reasons for this behavior are (i) trace linearization inside the code cache (of the binary translator) for the translated standard libc function and (ii) the inlining of all system calls from the `ld-linux.so` library.

The DynaRace prototype implementation uses multiple system calls to verify that already accessed files have not changed between system calls. For new files additional system calls are used to

gather information for the mapping cache. This setup leads to an overhead for file-based system calls of around 3-4x (for raw system call performance) to remove potential race conditions. Only the small set of file-based system calls that check or modify file metadata incur overhead. The overhead can be reduced through future optimizations that reuse more information or keep frequently accessed files open. All other (non file-related) system calls do not incur overhead.

The overhead is tolerable as system calls used to check the metadata of files and to open/close files are rare compared to read or write operations or computation. There is no overhead in the access to a file's data (e.g., reading from a file or writing to a file). Any overhead caused by DynaRace is associated with file metadata management. For most programs system calls that modify file metadata are not on the hot path. The time spent for I/O dominates the time spent for metadata management. The additional checks' overhead is small compared to the cost of a potential exploit.

6.2 Apache web server study

This section presents an end-to-end performance evaluation using the Apache web server. The web server study compares the performance of a system with active DynaRace protection for the Apache server and all scripts to a system without DynaRace. This study shows the completeness and end-to-end performance of our prototype implementation. The study uses Apache 2.2 on 32bit Ubuntu 10.04 LTS on a Core i7 950 CPU at 3.07 GHz in a VirtualBox virtual appliance using a single core. Apache uses the default Ubuntu configuration (multiple processes multiple threads, full support for the dynamic PHP interpreter and other modules). The study uses the `ab`[3] Apache benchmark in a different virtual machine on a different core on the same CPU to download four different files from a server. Each file is downloaded 100'000 times and Apache is restarted between iterations.

The four different files are `index.html`, a 5kB HTML file; `picture.png`, a 1mB image; `test.php`, a short PHP script that generates 90B output; and `test2.php`, a PHP script that executes `phpinfo()` and generates 48kB of output.

The evaluation uses three different system configurations: (i) native, Apache runs unmodified and unprotected; (ii) libdetox, Apache runs in a protected environment under the control of libdetox[4]; (iii) DynaRace, Apache runs in a protected libdetox environment with a DynaRace module that protects all file accesses as well. All three configurations run inside a virtual machine.

	Native	**libdetox**	(ovhd.)	**DynaRace**	(ovhd.)
index.html	1464	1675	-14.5%	1601	-9.4%
image.png	48	51	-6.3%	47	1.6%
test.php	1773	1562	12%	1498	15.5%
test2.php	463	343	26%	320	30.9%

Table 4. Results for the Apache benchmark showing the number of requests per second. The benchmark uses four different files to compare native performance, libdetox performance, and DynaRace performance.

Table 4 shows the performance numbers of the Apache benchmark. The three different system configurations are evaluated for each file. Each column shows the number of completed requests per second, the libdetox and DynaRace columns also show the overhead compared to native performance. The relative overhead shows that libdetox and DynaRace exhibit a small performance improvement compared to the native performance for static files. The improvement comes from the binary translation that results in greedy trace extraction for hot code.

For the dynamic files (`test.php` and `test2.php`) libdetox results in 12% and 26% overhead due to the translation of the dynamic PHP interpreter (which results in many indirect control flow transfers). Comparing the libdetox and DynaRace columns shows that DynaRace results in roughly 6% more overhead than libdetox alone. This overhead comes from the additional system calls for the directory and file atom authentication.

The overhead of both libdetox and DynaRace is tolerable, especially when considering that only a prototype implementation is evaluated. The prototype implementation still leaves room for additional performance optimization (e.g., additional caching of open file descriptors, better code optimization, and other optimizations). The current prototype implementation shows that the DynaRace approach is feasible and the overhead is tolerable. Additional optimization is left as a topic for future work.

6.3 Evaluation of the protection for file-based race conditions

DynaRace relies on the handler functions for the different system calls and the state of each file to protect and to remove potential race conditions. The file state is updated using transitions from one state to another.

This section shows three important usage scenarios that are common in applications. The usage scenarios are (i) checked file access where access permissions for a file are checked before it is used, (ii) temporary file creation where a file is created in an unsafe directory, and (iii) log rotation where files are replaced by an external process. Each usage scenario is dissected into the individual system calls and what (implicit) assumptions are used between the system calls. The described usage scenarios are prone to race conditions in their original form. DynaRace adds additional state to each file and enforces the assumptions between the system calls.

State transitions between system calls are shown in the following way:

$$oldstate \xrightarrow[targetstate]{systemcall} newstate$$

The state associated with a file transitions from *oldstate* to *newstate* if a *systemcall* system call is executed that induces the *targetstate* state. Both *newstate* and *targetstate* are needed for files that are in the *enforce* state. If a system call would induce the *update* state (e.g., the `access` system call) and the file is currently in the *enforce* state then the file remains in the *enforce* state.

6.3.1 Checked file access

The checked file access pattern consists of four steps to work with a file. The first step checks access permissions of the existing file using the `access` (or `stat`) system call. The second step opens the same file using the `open` system call, returning a file descriptor. The third step uses the file (e.g., reading from, or writing to the file descriptor using the `read` and `write` system calls; `read` and `write` system calls do not change the state of the file). The fourth step closes the file again using the `close` system call, completing the usage pattern.

A potential attack races to exchange the checked file after the first validation step and before the second opening step. Using mazes of connected directories [6] an attacker can win this race every time if he or she can inject a symbolic link into any part of the file name (see Section 7.4).

With DynaRace the following four state transition sequences are possible:

[3] The command used for the measurements is: `ab -c 2 -n 100000 http://${VM}/${FILE}`.

[4] Libdetox [21] protects applications from code-injection exploits, return oriented programming [26], and other control-flow oriented attacks

The file has not been accessed/used before by the application:

$$new \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \quad (1)$$

The file has been used before but was closed by the application:

$$retire \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \quad (2)$$

The file has been accessed by the application:

$$update \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} retire \quad (3)$$

The file is still opened by the application:

$$enforce \xrightarrow[update]{access} enforce \xrightarrow[enforce]{open} enforce \xrightarrow[retire]{close} enforce \quad (4)$$

The above state transition sequences leave no opportunity for a potential attacker to change files between a check system call and a "use" system call. An attacker could swap files in the *update* or *retire* state but not if the current state or next state is *enforced*.

A potential attacker tries to change a file after the `access` system call and before the `open` system call. Using the DynaRace states an attacker can change the file only before the `access` system call in Equation 1; in Equation 2 and Equation 3 a warning is printed, and in Equation 4 the application is terminated with a race condition error message. DynaRace guarantees that in all cases no unchecked file can be injected between the `access` and the `open` system call. A potential attacker can change the files before the permissions are checked but never between the permission check and the system call that uses the checked file.

As soon as a file transitions into the *enforce* state the last checked or accessed information is fixed and verified. This setup results in a guarantee to the program that the last checked information of a file is enforced when that information is used later.

6.3.2 Temporary file creation

Safe temporary file creation is an important and hard problem. An access test using an `access` or `stat` system call first checks that the file does not exist. A following `creat` system call (or an `open` system call with the create exclusive flag) then creates the file. An attacker can try to race between the existence check and the file creation to add a link to an already existing file that will then be overwritten.

$$new \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \quad (5)$$

Equation 5 shows the state transitions when a new file is created. The state information contains a valid directory and the error code for the file after the `access` or `stat` system call. The directory where the file will be placed in must exist but the temporary file must not exist.

6.3.3 Log rotation

Log rotation is a technique that is used by many daemons to renew/rotate their log files. A daemon opens a log file and writes information to that file. A rotation daemon moves these log files to a different place and signals the original daemon to reopen the log file. The daemon then closes the old file and reopens the log file (which no longer points to the same inode and device id).

DynaRace's *retire* state enables log rotation. When a file is closed it can be reopened even if the metadata of the file has changed. The *retire* state is similar to the *new* state where we do not know anything about the file. The information in the cache is updated and enforced only if there exists a valid set in the cache.

```
...
lfd = open(tmp, O_CREAT|O_EXCL|O_WRONLY,
           0644);
...
if (lfd < 0) {
  unlink(tmp);
}
...
write(lfd, pid_str, 11);
/* unchecked relaxation */
chmod(tmp, 0444);
...
```

Listing 2. TOCTTOU vulnerability in X.org (os/utils.c, C code).

$$enforce \xrightarrow[retire]{close} retire \xrightarrow[update]{access} update \xrightarrow[enforce]{open} enforce \quad (6)$$

Equation 6 shows a sequence of system calls that is used during the rotation of a log file. The file is closed by the daemon, checked, and reopened. The `access` system call following the `close` system call prints a warning message that the file has changed. The daemon continues in the *update* state and enforces the information from the check when the file is opened.

This scheme leaves a window of opportunity for the attacker where he or she can change the log file after it was closed but before it is accessed. The access check in the daemon catches these attacks and the state information is later enforced.

6.4 X.org file permission change vulnerability

Version 1.4 to 1.11.2 of the X.Org X11 X server had a severe TOCTTOU race condition [34]. The X server creates a temporary lock file and relaxes the permissions of the lock file using unsafe system calls. Any local attacker with permission to run the X server can exploit this vulnerability to set the read permission for any file or directory on the system.

Listing 2 shows the code containing the race condition. The variable `tmp` contains a fixed string of `/tmp/.tXn-lock` where X is the n-th X display running on that computer. An attacker executes the SUID X binary as P_1. The attacker stops P_1 after the `write` system call and before the `chmod` system call. In a second execution P_2 of the X binary the `open` system call fails and the temporary file created by P_1 is removed using the `unlink` system call. The attacker kills P_2 and links `/tmp/.tXn-lock` to an arbitrary file (e.g., `/etc/shadow`) and continues P_1. P_1 will then set the arbitrary file to world-readable.

DynaRace would protect from this TOCTTOU race condition. As soon as the file identified by the `tmp` variable is in the *enforce* state it can no longer be modified by a concurrent process. Equation 7 shows the states for the temporary file for P_1. Before the `chmod` system call P_1 is stopped and P_2 depicted in Equation 8 is executed. P_1 then continues and DynaRace throws an error because the metadata of the enforced temporary file has changed.

$$P_1 : new \xrightarrow[enforce]{open} enforce \xrightarrow[enforce]{chmod} FAIL \quad (7)$$

$$P_2 : new \xrightarrow[enforce]{open} enforce \xrightarrow[unlink]{retire} retire \quad (8)$$

7. Related work

DynaRace is a dynamic TOCTTOU race detection mechanism that is implemented as a libdetox extension. File-based race detection

can be built on different techniques. Static race detection analyzes the program source code or the binary of the application before execution; dynamic race detection uses code that is added during the compilation of the program to detect races at runtime; dynamic race prevention uses added code to prevent possible races; and novel race-free APIs change the available file system API to remove or reduce potential races.

7.1 Static race detection

Static source-code analysis [10, 11, 23, 33] can be used to detect potential file-based race conditions. A scanner reads the source-code of the application and observes calls to library functions or system calls. If a sequence of calls opens a race condition then the static analysis tool emits a warning.

Static analysis tools often have a high number of false positives because they rely on pattern matching and they cannot use runtime data to verify potential races. Due to the high error rate these approaches are not feasible for online services that must meet some response time constraints.

The DynaRace approach adds state to each accessed file. Due to the classification of system calls into groups and the additional state DynaRace decides for each system call if a potential race condition occurs.

7.2 Dynamic race detection

A dynamic approach can observe all system calls as they happen. These system calls can either be logged and analyzed post-mortem [18] or an online analysis can evaluate the pair of system calls according to a given policy [28]. Aggarwal and Jalote [3] use static binary translation to add dynamic guards that are executed at runtime.

IntroVirt [16] uses OS virtualization and a virtual machine monitor to run predicates outside of the OS scope. These predicates check the health of system software and also check for file race conditions if the programs have the right predicates. Wei and Pu [35] enumerate likely TOCTTOU pairs that are used for common tasks. Goyal [15] uses the `strace` framework to collect system call traces and detect TOCTTOU attacks.

The technique of enumerating pairs of system calls misses the opportunity to detect race conditions dynamically between any combination of system calls. This approach is similar to blacklisting several pairs of system calls without looking at all possible combinations. Only an approach like DynaRace that takes care of all file-based system calls can protect from all race conditions. Due to the added state DynaRace looks at all possible combinations and sequences of system calls and therefore provides complete coverage.

7.3 Dynamic race prevention

RaceGuard [12] defends against several classes of TOCTTOU attacks using an in-kernel cache but is limited for file swap attacks where the file itself is changed from one execution to the other. Tsyrklevich and Yee [31] implement pseudo-transactions in a kernel module to detect specific combinations of check/use system calls targeting the same file. Multiple similar approaches followed [20, 27, 32, 36].

Several novel race-free APIs have been proposed to replace the POSIX-like system calls for file handling. Some of these novel APIs use transactions to run a set of related system calls atomically [22, 37]. The OS insures that no other application may interfere with these system calls.

Dean and Hu prove that no deterministic solution to prevent races exists (with the set of available system calls in 2004) and introduce a probabilistic way to prevent races [13]. They rely on the assumption that an attack is unable to win all races. Therefore they propose hardness amplification. Hardness amplification executes the check multiple times in a loop (a so called K-loop) and checks if all results are as expected.

Transactions of system calls need kernel modification. In addition to the new code in the kernel the applications must be rewritten to include transactional code. DynaRace supports a gradual approach that adds protection from file-based TOCTTOU races. Applications and libraries can use a mix of old file-based system calls and new file-based system calls. Using the file-state and the new system calls DynaRace deterministically protects every file-based system call.

7.4 Maze-based race attacks

Borisov et al. break the probabilistic approach by enabling an attacker to win all races [6]. If all races are won by the attacker then the check proposed by Dean and Hu [13] is unable to detect the race condition. An attacker constructs so called mazes. A *maze* is a chain of temporary directories that contains a link to the next chain as the leaf. The attacked program needs time to go through the maze (e.g, once for `access` and once for `open`) and in that time the attacker swaps the final link.

7.5 Hardness amplification-based race protection

Mazières and Kaashoek [19] propose an alternate approach that uses inodes (low level file objects) instead of filenames. If the binding between inode and filename is immutable then no file swap race conditions are possible when the target file changes. Tsafrir et al. [29, 30] implement a similar approach in user-space. Their approaches render the maze attack from Borisov et al. [6] impossible. User-space checks guard applications from TOCTTOU attacks. A file is resolved step by step. The algorithm checks for each part of the filename if the current part is a file or a symlink. Files are opened using an approach similar to Dean and Hu [13] using a K-loop probabilistic check. Symlinks are opened using a recursive check using the same algorithm.

Chari et al. take a different approach and extend Tsafrir's work to implement a `safe-open` mechanism [9] that ensures that all elements of a path are safe to open by the current user id. Safe file elements can be modified only by the same user or root.

DynaRace introduces a hidden mapping and state cache and rewrites unsafe system calls. This technique gives similar guarantees as Mazières and Kaashoek without the need to change the implementation of the application or the libraries. DynaRace resolves a file step by step similar to Tsafrir et al. but uses safe system calls. Using the new system calls DynaRace is deterministic and does not rely on a probabilistic approach.

7.6 System call interposition in binary translators

System call interposition adds an intermediate layer between the application and the operating system. This intermediate layer intercepts every single system call and enables checks based on the system call, the parameters, and the state of the application.

Binary translation is a natural solution to this problem as it translates the running program and can intercept any instruction or any memory location. Static binary translation is limited to statically known code, dynamic binary translation enables loadable modules as well. Dynamic binary translation can be used to weave additional security guards into the running application by modifying the executed machine code with low overhead. libdetox [21], Vx32 [14], Strata [24, 25], Program shepherding [17], and DynamoRIO [7, 8] enable low overhead dynamic binary translation.

We choose libdetox for our implementation because it contains many additional security guards that protect the running application

from other attack vectors like buffer overflows, code injection, and (to some extent) data attacks.

8. Limitations and weaknesses

This section discusses limitations and weaknesses of the DynaRace approach and highlights insights obtained through the DynaRace prototype implementation. The idea of DynaRace is to associate state to each file. Files are identified using a mapping cache that maps filenames to unique inodes and device numbers.

Using the state machine in Figure 3 enables specific checks for file-based system calls, whereas the state transitions are decided based on the group that the current system call is in and the state of the file. All usage cases of a file inside an application are covered using the state diagram that never throws an exception if the file has not been changed between different system calls.

The *retire* state enables multiple concurrent processes to modify files in a safe way. Closing a file in one application retires the information in the mapping cache and the application no longer assumes that the state of the file remains the same. Ownership transfers in, e.g., log rotation, are only possible using the special *retire* state.

8.1 Retirement schemes

The current implementation keeps all files in the *enforce* state after they have been used with a system call of the *use* group. A possible extension of the state machine in Figure 3 would be a retirement scheme for system calls that change the file but do not return an open file descriptor (e.g., `chmod`, `chown`, or more general system calls in the *use* group without `creat` and `open`).

These system calls do not return an open file descriptor when they are executed. The application does not signal the kernel that it no longer expects to use the associated file (e.g., through the `close` system call for opened files). DynaRace therefore cannot check if a file is no longer used for these system calls. The retirement scheme for these system calls could be implemented through a timer that starts when the system call is executed. The timer would signal when these files can be retired and the state of the file would move from *enforce* back to *update*, thereby enabling concurrent modification without terminating the program.

A timer is of course a dangerous addition as it opens a new windows of opportunity for an attacker to delay the application until the timer runs out and the file state is no longer enforced. We leave this problem to future work.

8.2 Broken (legal) usage scenarios

DynaRace protects from changes to file-metadata from concurrent processes if the file state is enforced (i.e., if the application assumes that information from prior system calls is still valid). Possible broken usage scenarios are concurrent file/directory modification and polling.

If an application works with a file in the *enforce* state and a concurrent process changes the parent directory (e.g., by moving the directory, or by renaming the directory) then the directory verification will fail and an error is thrown. Two concurrent processes also cannot work together on a single file if it is in the enforce state in both processes.

Polling (i.e., checking metadata of the file) from concurrent processes works as long as the file is in the *update* state. If both processes want to work with the file (e.g., reading, writing, changing metadata), then DynaRace will terminate the second application due to a metadata mismatch.

DynaRace limits the number of modifiers of each file to a single process. If multiple processes try to modify a file at the same time then all processes except the first are terminated due to metadata mismatches.

9. Conclusion

This paper presents DynaRace, a novel approach to detect and prevent file system races in unmodified applications. DynaRace adds state to each file and keeps metadata for all accessed files in a mapping cache.

File system races are detected if the information in the mapping cache does not match the information of the current file. Our prototype application emits a warning and terminates the application if races are detected.

DynaRace ensures that there are no file-based races possible and dynamically rewrites unsafe system calls into safe versions that are race-free. A benefit of the DynaRace approach is that it protects all file-based system calls. DynaRace allows partial migration of individual libraries to the new set of file-related system calls. The new system calls are no panacea, even with new system calls a programmer still has to worry about the state of individual files. DynaRace removes this burden from the programmer and offers a state-based dynamic approach for all file-related system calls.

Currently programmers have to live with potential races when they use standard system calls, and it is hard to program race-free applications using the new system calls. The DynaRace approach enables programmers to use both the unsafe and the new system calls without the need to add explicit additional safety checks in the application while closing the door to attacks that attempt to exploit file system races.

10. Acknowledgments

We thank the anonymous reviewers for their detailed feedback and suggestions on how to improve the paper. Special thanks for pointing out [30].

References

[1] New system calls. https://lwn.net/Articles/164887/.

[2] openat syscall. http://linux.die.net/man/2/openat.

[3] AGGARWAL, A., AND JALOTE, P. Monitoring the security health of software systems. In *ISSRE'06: 17th Int'l Symp. Software Reliability Engineering* (nov. 2006), pp. 146 –158.

[4] BISHOP, M. Checking for race conditions in file accesses. Tech. rep., University of California at Davis, 1995.

[5] BISHOP, M., AND DILGER, M. Checking for race conditions in file accesses. *Journal for Computing Systems* (1996), 131–152.

[6] BORISOV, N., JOHNSON, R., SASTRY, N., AND WAGNER, D. Fixing races for fun and profit: how to abuse atime. In *14th USENIX Security Symposium* (2005), pp. 303–314.

[7] BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. Design and implementation of a dynamic optimization framework for Windows. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-4)* (2001).

[8] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (2003), pp. 265–275.

[9] CHARI, S., HALEVI, S., AND VENEMA, W. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS* (2010).

[10] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *CCS'02: Proc. 9th ACM Conf. Computer and Communications Security* (2002), pp. 235–244.

The source code of the DynaRace framework and additional examples can be downloaded at `http://nebelwelt.net/projects/libdetox`.

[11] CHESS, B. V. Improving computer security using extended static checking. In *S&P'02: IEEE Symp. on Security and Privacy* (2002).

[12] COWAN, C., BEATTIE, S., WRIGHT, C., AND KROAH-HARTMAN, G. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proc. 10th USENIX Security Symposium* (2001), p. 12.

[13] DEAN, D., AND HU, A. J. Fixing races for fun and profit: how to use access(2). In *Proc. 13th USENIX Security Symposium* (2004), SSYM'04, pp. 14–14.

[14] FORD, B., AND COX, R. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference* (2008), pp. 293–306.

[15] GOYAL, B., SITARAMAN, S., AND VENKATESAN, S. A unified approach to detect binding based race condition attacks. In *CANS'03: Intl. Workshop on Cryptology & Network Security* (2003).

[16] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP'05: Proc. 20th ACM Symposium on Operating Systems Principles* (2005), pp. 91–104.

[17] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proc. 11th USENIX Security Symposium* (2002), pp. 191–206.

[18] KO, C., AND REDMOND, T. Noninterference and intrusion detection. In *S&P'02: Proc. 2002 IEEE Symposium on Security and Privacy* (2002), pp. 177–187.

[19] MAZIÈRES, D., AND KAASHOEK, M. F. Secure applications need flexible operating systems. In *HotOS'07: Workshop on Hot Topics in Operating Systems* (1997), pp. 56–61.

[20] PARK, J., LEE, G., LEE, S., AND KIM, D.-K. RPS: An extension of reference monitor to prevent race-attacks. In *PCM'04: 5th Pacific Rim Conf. on Multimedia* (2004), pp. 556–563.

[21] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th ACM SIGPLAN/SIGOPS Int'l conf. Virtual execution environments* (2011), pp. 157–168.

[22] SCHMUCK, F., AND WYLIE, J. Experience with transactions in quicksilver. In *SOSP'09: Proc. 13th ACM Symposium on Operating Systems Principles* (1991), pp. 239–253.

[23] SCHWARZ, B., CHEN, H., WAGNER, D., LIN, J., TU, W., MORRISON, G., AND WEST, J. Model checking an entire Linux distribution for security violations. In *Proc 21st Computer Security Applications Conference* (2005), pp. 13–22.

[24] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. Tech. rep., University of Virginia, 2001.

[25] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. *ACSAC'02: Annual Computer Security Applications Conference* (2002), 209.

[26] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th ACM conf. Computer and Communications Security* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[27] SPILLANE, R. P., GAIKWAD, S., CHINNI, M., ZADOK, E., AND WRIGHT, C. P. Enabling transactional file access via lightweight kernel extensions. In *FAST'09: Proc. 7th conf. on File and storage technologies* (2009), pp. 29–42.

[28] SUK LHEE, K., AND CHAPIN, S. J. Detection of file-based race conditions. *Int'l Journal Information Security 4*, 1-2 (2005), 105–119.

[29] TSAFRIR, D., HERTZ, T., WAGNER, D., AND DA SILVA, D. Portably solving file TOCTTOU races with hardness amplification. In *FAST'08: Proc. 6th USENIX Conf. on File and Storage Technologies* (2008), pp. 13:1–13:18.

[30] TSAFRIR, D., HERTZ, T., WAGNER, D., AND DA SILVA, D. Portably preventing file race attacks with user-mode path resolution. Tech. Rep. RC24572, IBM T. J. Watson Research Center, June 2008.

[31] TSYRKLEVICH, E., AND YEE, B. Dynamic detection and prevention of race conditions in file accesses. In *Proc. 12th USENIX Security Symposium* (2003), pp. 243–255.

[32] UPPULURI, P., JOSHI, U., AND RAY, A. Preventing race condition attacks on file-systems. In *SAC'05: Proc. ACM Symposium on Applied computing* (2005), SAC '05, pp. 346–353.

[33] VIEGA, J., BLOCH, J., KOHNO, T., AND MCGRAW, G. ITS4: a static vulnerability scanner for C and C++ code. In *ACSAC'00: Ann. Comput. Security Applications Conf.* (2000).

[34] VLADZ. Xorg file permission change vulnerability (CVE-2011-4029). http://vladz.devzero.fr/Xorg-CVE-2011-4029.txt.

[35] WEI, J., AND PU, C. TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study. In *FAST'05: Proc. 4th conf. USENIX Conf. File and Storage Technologies* (2005), pp. 12–12.

[36] WEI, J., AND PU, C. A methodical defense against TOCTTOU attacks: the EDGI approach. In *ISSSE'06: IEEE Int'l Symp. on Secure Software Engineering* (2006).

[37] WRIGHT, C. P., SPILLANE, R., SIVATHANU, G., AND ZADOK, E. Extending ACID semantics to the file system. *Trans. Storage 3* (June 2007).