Summer 2009

Runtime Automated Detection of Out of Process Resource Management in the X Windowing System

Caolan McNamara
Regis University

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

Runtime Automated Detection of Out of Process Resource Mismanagement in the X Windowing System

Caolán McNamara

Regis University

School for Professional Studies

Master of Science in Software and Information Systems

Dedication

Martin Mellody 1975-2009

Table of Contents

Abstract

Software applications typically allocate and deallocate resources during their lifetime. Resources can be categorized into two broad groups, in-process and out-of-process resources where in-process resources are local resources directly managed by a client, while out-of-process resources are remotely managed by a client which instructs a server to allocate and deallocate the resource on its behalf.

Out-of-process resources do not reside in a clients address space which poses an extra layer of complexity in attempting to debug their misuse.

This thesis presents an automatic run-time solution to the problem of detecting and reporting source code locations of application client mismanagement of out-of-process resources for a specific case-study of the X Windowing System which lends itself to use in the wider general case.

Runtime Automated Detection of Out of Process Resource Mismanagement in the X Windowing System

When Software applications allocate and deallocate resources during their lifetime it is common for programmers to accidentally:

1. Fail to deallocate resources after use has been completed
2. Attempt to re-use a resource that has been deallocated
3. Attempt to use a resource that has not yet been allocated

Attempting to re-use a deallocated resource, or an unallocated resource generally results in some type of failure of the flawed software. Failure to deallocate resources causes resource leaks; over time these resource leaks can starve the system of available resources leading eventually to failure in either the afflicted software, or another application attempting to gain sufficient resources to function. On modern operating systems most resources in use by an application are released on exit, but long-lived applications such as web-browsers or office-suites can accumulate enough leaked resources over their life-time to noticeable degrade the overall system performance.

The misuse of in-process resources such as memory and file handles is well documented and understood (Dumitran, 2007), (Maebe, Ronsse, De Bosschere, 2004). A number of programmers' tools exist to detect when a handle to an in-process resource was lost without first deallocating the resource, or when an operation has been attempted on an invalid handle. Such tools can display the location within the source code where this has occurred. Some tools can also additionally track the use of resources over the life-time of a process, and also report the location where an invalid resource handle became invalid, or where a leaked handle was originally allocated.

Problem Statement

The problem of out-of-process resources is similar , but one with an extra layer of complexity in that client software instructs a server to allocate or deallocate a resource on the client's behalf rather than making a direct in-progress allocation or deallocation. Analogous to in-process resources, out-of-process resources are controlled by the client but differ in that the resources do not reside in the address space of the client process. While they are typically deallocated by the server on loss of connection of the client, long lived clients can cause the same type of resource leaks for server resources as can happen with in-process resources. Attempting to use an unallocated or released resource may cause the server to report the error to the client, or to terminate the client, but errors may be reported asynchronously, i.e. the application may not be informed immediately after use of an invalid handle that it was invalid but instead at some later stage, making it more difficult to associate the error with the location that triggered the error.

To resolve these problems the client-side programmer needs detailed and reliable information which is relevant to detecting and solving these out-of-process resource errors, i.e. the source-code location within the program where the initial error was introduced and where the error was manifested.

Purpose of Thesis

This thesis presents a solution to detecting and reporting the source code locations of misuse of out-of-process resources. To determine and implement as a programmers' aid for a specific case study of the X Windowing System, a mechanism for detecting and locating when and where handles to the server-side resources have:

1. been lost without a directive to instruct the server to deallocate the associated resource
2. been used after a deallocation directive

3. been used without an allocation directive

The techniques deployed in this tooling can be applied to the wider general case of generic client-side analysis of client-side controlled resources which exist in an out-of-process server.

Case Study

The X Windowing System is one method for providing a Graphical User Interface (GUI) s on UNIX and UNIX-like operating systems. It is an example of a client-server architecture where resources are allocated by a server on instruction by clients, which may or may not be on the same machine as the server. Consequently, X resource leaks and misuse are difficult to identify and locate (McCullagh, 2008). Tools exist (Allum, 2003) to indicate that an application is exhibiting suspicious resource growth which likely indicates the presence of a leak, but not to identify where within the client source code that this potential leak occurs, nor to report on any other class of resource misuse.

Assumptions and Goals

The basic assumption is that the tool must be deployed on the client-side rather than server-side in order to supply client-side source-code locations through use of the debugging symbols of the client application binary.

The other major assumption is that the tooling should require no modification of the application itself. As motivation for tooling which requires no modification of the application to be debugged an example target application which could benefit from such analysis is OpenOffice.org, which on contemporary hardware requires approximately 5 to 6 hours to build. Requiring a complete rebuild in order to instrument it to enable detection of out-of-process resource errors would be an inordinate up-front burden on the programmer.

The goal is a successfully implemented mechanism which is capable of operating on

unmodified real-world large application binaries such as those of the OpenOffice.org office suite or Firefox web browser and accurately report source-level locations of X resource client-side mismanagement during runtime.

Exploring the problem space

Overview of the X Window System

The X Window System, a trademark of The Open Group, is a client/server architecture where multiple client applications connect to an X server (Gettys & Scheifler, 2002). The applications are the clients, they communicate with a X server which controls the physical graphic display. Clients issue requests to the server which executes them on the clients behalf, e.g. drawing requests, window creation, window destruction etc, while the X server relays user interaction events to the client, e.g. mouse clicks, keyboard events, etc. (Manrique, 2001).

The term X Window System does not indicate any specific product or implementation, but instead is defined by The Open Group as a set of protocol and application programmer interface (API) specifications. A X server is not specified beyond the X Protocol which defines the structure of the data which is shipped to and from that server. The X Window System been implemented by multiple vendors to create multiple interoperable implementations. In this study the implementation used was the XOrg Foundations's Open Source public implementation, though no non-standard features of this implementation were used which do not exist in all other implementations of the X Window System provided by other vendors.

The crucial architectural feature of the X model is that it doesn't constrain the client to execute on the same machine as the server, the communication protocol can work over a network as well as over a local inter-process channel. In either local or remote case the client and server operate in different address spaces, and communicate over a serialized protocol, rather than execute in the same local address space.

The X Window System specifies a C subroutine library, named Xlib, which supplies a base layer API for drawing and windowing operations. Applications link against Xlib, issue direct in-

process calls to the Xlib API, and Xlib takes care of converting those API calls into the underlying X Protocol which is shipped across to the out-of-process X server though some communication pathway hidden to the client.

As an example, the following illustration shows multiple client applications making use of Xlib's XDrawArc function which Xlib converts to the X Protocol and ships it over the network to a remote server which renders the arcs to the screen it controls.

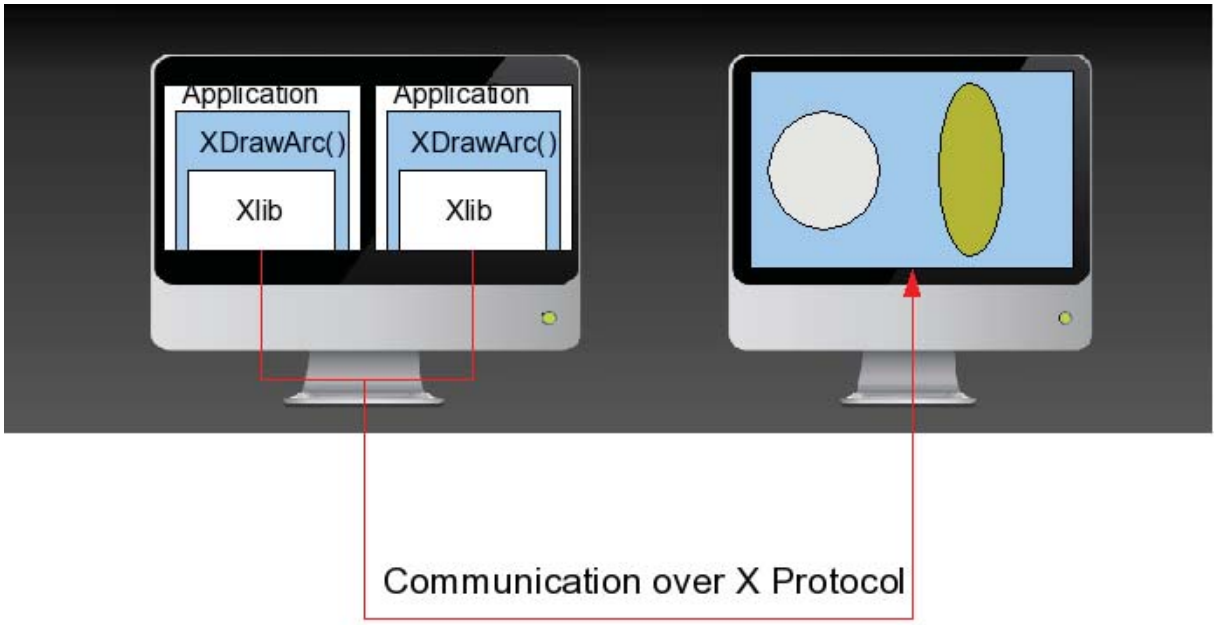

Illustration 1: X Window Architecture

Remote X Resources

The problem to be solved is to diagnose on the client-side, the misuse of remote resources controlled by the client that exist in the X server. At the client-side, these resources are identified by a simple integer number, i.e. “many Xlib functions will return an integer resource ID, which ... refer to objects stored on the X server” (Gettys & Scheifler, 2002). These remote resources which are controlled by the local client via these integer handles can be of type Window, Pixmap, Cursor, Font and Colormap.

Window: A window is a region of the screen which can be shown or hidden (mapped or unmapped).

Pixmap: “An off-screen graphics object. Pixmaps can be used in most graphics functions interchangeably with windows and are used in various graphics operations to define patterns or tiles” (Gettys & Scheifler, 2002). A Pixmap can be copied to a window, so Pixmaps often used in double-buffering to rapidly update a Window without repeating a series of drawing operations.

Cursor: An image that is shown for a mouse pointer, as opposed to a text entry caret that indicates the current text insertion point.

Font: These server side fonts are considered deprecated in favour of client-side fonts, but their use is still supported. Server-side fonts reduced the amount of data that must be transmitted from client to server, but limited clients to the fonts available on the server (Herrb & Hopf, 2005).

Colormap: “The colormap is a small table with entries specifying the RGB values of the currently available colors” (Lee, 1992). Colormaps are of most use in 8 or 16 bit displays where the number of colours that could be shown at one time is limited. Their use with more common 24 and 32 bit contemporary displays is less of a factor than historically

Window and Pixmaps are collectively known as Drawables and are often interchangeable for various graphic operations. A major difference is that Windows are always in a hierarchy while Pixmaps are not. Every application Window has a parent, and destroying a parent automatically destroys all children of that parent. The Xlib API to create a Pixmap requires an existing Drawable to be provided but the resulting Pixmap is not a child of that Drawable and not placed in a hierarchy. A Pixmap is not destroyed when the reference Drawable is destroyed.

Given the deprecated state of server-side fonts and the increasing unlikelihood of a need to use Colormaps the most commonly used server side resources are Pixmaps, Windows and Cursors. Of these Pixmaps are inherently the easiest to allow to leak or otherwise misuse. They are off-

screen so failing to release them has obvious visual effect and as Drawables their similarly to Windows might erroneously suggest that they are destroyed automatically when the reference Drawable used to create them is itself destroyed. The server-side footprint of a Pixmap varies according to their dimensions and colour depth, and large numbers of pixmaps can consume significant amount of server-side memory. Pixmaps leaks has been shown to cause very serious resource starvation in X applications (Giraldeau, Dault & des Ligneris. 2006, McCullagh, 2008 & Erikson, 2009) leading to an inability of the X Server to provide new Pixmaps to any clients.

Implementation Note

Destroying a window automatically destroys all child Windows, this differs from the destruction of other resources. In order to correctly report on Windows which were created but not destroyed an implementation will have to capture the hierarchical relationship between windows in order to flag children of a destroyed parent as themselves destroyed.

Problem Summary

Resources appear on the client side as integer ids. Those integer resource handles are provided to an application as the result of calls through the Xlib library which communicates to a out-of-process X server. There are five classes of resource handles, one associated with a hierarchy where destruction of a parent results in destruction of children.

Existing Technology

X Window Resource Usage Technology

Xrestop: “Xrestop uses the X-Resource extension to provide 'top' like statistics of each connected X11 client's server side resource usage. It is intended as a developer tool to aid more efficient server resource usage and debug server side leakage” (Allum, 2003).

Some sample output is shown below

```
xrestop - Display: localhost:0
          Monitoring 40 clients. XErrors: 0
          Pixmaps:   81195K total, Other:     181K total, All:   81376K total

res-base Wins  GCs Fnts Pxms Misc   Pxm mem  Other   Total   PID Identifier
4600000    81  175    1  806  149    25856K    10K  25867K 10413 OpenOffice.org Impress
1c00000  1376   67    1   54   80     3608K    36K   3645K  2390 gtk-window-decorator
4000000   532  307    1  190  542     2868K    33K   2901K  2671 Graphics - Mozilla Firefox
0e00000    26   39    0   18   76      169K     3K    172K  2415 wnck-applet
4c00000    28   50    1   16   45        87K     3K     91K 11328 xlib.pdf
2c00000    12   39    0   14   35        3K     2K      5K  2357 Evolution Mail and Calendar
1200000    43   47    0   20   56        1K     3K      5K  2310 Panel
1000000     6   28    0    2  176        8B     4K      4K  2302 gnome-settings-daemon
3a00000     6   28    1    1   20        4B     2K      2K  2493 tomboy
1600000    12   52    0    2   29        8B     2K      2K  2492 notification-area-applet
1a00000     4   28    0    2   34        8B     1K      1K  2347 gnome-power-manager
3800000     6   37    0    2   13        5B     1K      1K  2496 clock-applet
2a00000     2    3    0    2   47        5B     1K      1K  2373 notification-daemon
3200000     4   28    0    2   12        8B     1K      1K  2345 applet.py
2200000     4   28    0    2   12        8B     1K      1K  2348 Bluetooth Applet
2000000     4   28    0    2   12        8B     1K      1K  2346 NetworkManager Applet
1e00000     4   28    0    2   12        8B     1K      1K  2352 gnome-volume-control-applet
```

The X Resource Extension allows the quantity of each type of resource and the memory associated with them to be queried from the X server by a client. It provides access to the information known to the X server about resource utilization. It can be used to identify suspicious behaviour in an application which may indicate a resource leak, but it can only report what the X server knows, and the out-of-process X server does not, and can not, know where within the applications source code the resource leak may exist. For the same reason it does not, and is not intended to, report on use of deallocated or unallocated resouces.

The XRes lead developer Matthew Allum (2008) plans “future work involving event generation on resource creation/destruction”, which might provide some degree of X server-side

support for a speculative client-side debugging tool to be informed of these events. This further extension does not exist as of the time of writing, and conceptually there remains the difficultly on receipt of an event by a debugger to map these proposed, and possibly asynchronous, events back to the source code location within a client which indirectly triggered the creation/destruction event via the server.

Analysis and Debugging Technology

There is not a great deal of existing literature on the specific problem addressed by this thesis, but there is proven technology used to solve similar problems which provide insights and possible technological frameworks which could be adapted for use to implement a solution.

Dtrace: “DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs” (Sun Microsystems, 2009).

Dtrace has been shown (Cantrill, Shapiro & Leventhal, 2004) capable of being used to server-side dynamically instrument a running X server to detect unusual activity and isolate the individual connection from the offending client. And to then be used client-side to instrument that client and detect the Xlib library calls which are known to map to that server behaviour.

DTrace is a script-able framework available only for the Solaris operating system which can be used to query and report on a large number of kernel and user-level events that an application triggers without modifications to the application itself. As a toolkit it is possible to speculate that DTrace has sufficient features to be used to implement tooling which captures client-side Xlib function calls, examine their arguments and track what resources have been created, but not destroyed and identify use of deallocated/reallocated resources. But no such implementation is documented to exist. The tie to the Solaris operating system makes implementing a solution based

on DTrace equally limited to Solaris.

Purify: A commercial program that “developers and testers use to find memory leaks and access errors” (Hastings & Joyce 1992). Purify is a dynamic binary analysis tool that reports errors at run-time of the application being tested. Before execution the application is re-linked by purify in order to rewrite the binary to intercept attempts to read and write memory and track if an attempt to read/write is on an invalid or uninitialized area of memory.

Purify solves the analogous problem of detecting misuse of memory as an in-process resource and can report on memory leaks, but has no mechanism for extension nor is Purify's source available for modification to base an adaptation which could perform the same task for out-of-process resources.

DSO interposition: Programmers commonly block code together into libraries. Libraries whose code is bound to at run-time rather than at link time, and which can therefore be shared between multiple applications at the same time are termed shared libraries, or Dynamic Shared Objects (Drepper, 2006). The Xlib library is one such library. Among the features of a Dynamic Shared Object (DSO) is the capacity to override individual functions that an application would call from a DSO by providing at application launch-time another shared library with functions of the same signature as those found in the normal library. The dynamic linker can be trivially requested to resolve attempts to find dynamic symbols against the provided shared library before searching the standard libraries.

Another feature of a DSO is that there is an API to explicitly search for functions by name in a named shared library and bind function pointers to them. By combining the two techniques a shared library can be written which can be interposed between the application and the normal shared library. The interposed library can provide methods which override the standard library, carry out additional work, and forward the method onwards to the standard library.

DSO interposition is a generic technique which has been successfully applied to solving a wide range of similar problems, e.g. detecting and fixing file descriptor leaks (Dumitran, 2007) and profiling Xlib function calls (Curry, 1994).

Support for extracting the source code file and line number from within a shared library to determine where within the application the call originated is then available through the use of the backtrace function call provided by the Linux standard C library and mapping the resulting data with existing debug information tools (McNamara, 2007).

Valgrind: "A programmable framework for creating program supervision tools such as bug detectors and profilers. It executes supervised programs using dynamic binary translations, giving it total control over their every part ... without the need for recompilation or relinking prior to execution" (Nethercote & Seward 2003). Valgrind is a basis on which various execution analysis tools can be built. The best known tool is Memcheck which can detect: use of uninitialized memory, use of deallocated memory, use of unallocated memory and memory leaks. Unlike purify the source is available and modification is allowed. Valgrind is extensible and a number of diverse tools have been successfully implemented using the Valgrind core.

Valgrind, unlike the tracing framework Dtrace, and unlike other dynamic binary instrumentation frameworks such as ATOM (Rivastava & Eustace, 2004) or PIN (Luk, et.al, 2005), supports *origin tracking*. Origin tracking enables the location of where an invalid value was initially injected into the program flow and is the mechanism by which the Valgrind tool Memcheck implements identifying the line of code where an invalid pointer was initially assigned to a variable.

Without origin tracking, an analysis tool can report that an invalid value has been operated on, and show the immediate stack-trace at that point. The immediate history of where the value was passed down from is clear from a stack trace, but the history of propagation of the value from the point where it was initially assigned an invalid point to the entry point of the stack-trace is not

known. With origin tracking, the question of *why* a value is invalid can be answered by recording program locations where unusable values are assigned and storing this information in place of the unusable values themselves, facilitating the automatic support of propagating the origin information for an invalid value piggy-backed on the value itself has it propagates through the program flow, making it available at error detection time to report the origin of the invalid value.

Valgrind's extensible nature, proven real-world suite of tools based on it, powerful origin tracking and accessible documentation makes it a very attractive foundation for building program analysis tooling.

Solution Architectures

There are a number of possible approaches to solving the problem of tracking use of remote resources in order to report to the programmer the source-code location within the program of leaks and misuses of them. The specific capabilities required are the capabilities to report, without recompilation of the application, the source-code locations of:

1. where a leaking resource was allocated
2. where an invalid resource was operated on
3. where an invalid resource was previously deallocated or initially incorrectly allocated

This section illustrates the possible solution architectures, their individual strengths and weaknesses and examines their capabilities to fulfil the stated goals.

Common Features

The common feature of all approaches is the necessity to detect and capture the resource allocation, resource deallocation, and resource utilization events, and the source-code origin of those events. Each solution needs to record allocation and deallocation events, and to examine utilization events in order to compare the utilized resource against previously allocated and deallocated resources.

Resources are identified in a client by integer values, and are therefore basically indistinguishable from any other integer value used by the application. To track them the mechanism by which they enter into the application must be captured. There are two major options for capturing this information for the specific case study of the X Window System: at the X Protocol level where the information from the X server is received, or at the level of Xlib API entry points.

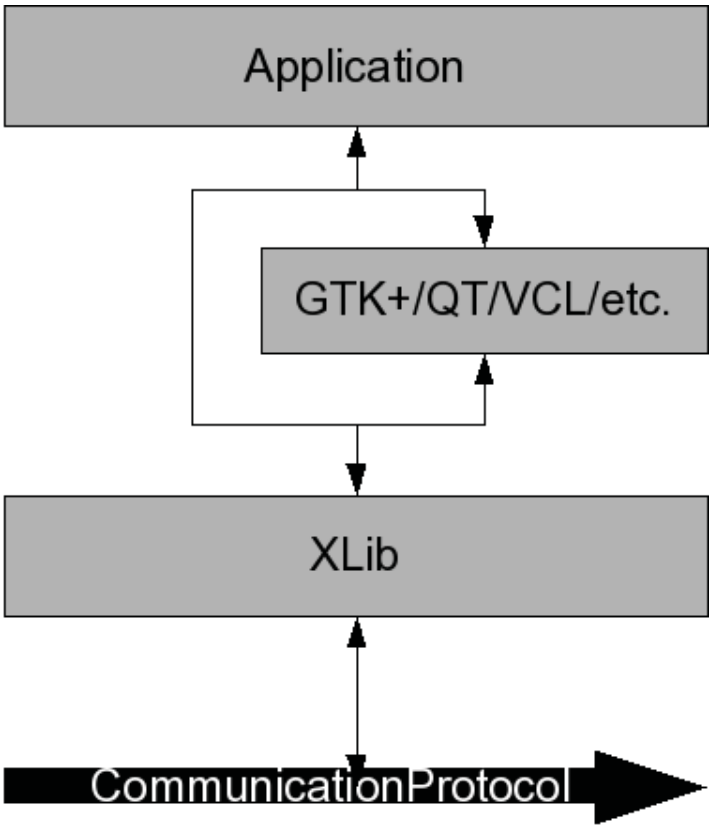

Illustration 2: Normal Application Stack

Capturing at entry point to Xlib library calls has the advantage that the same technique is applicable to a far wider set of similar problems where integer handles enter the application space through specific API function calls, while protocol level capture is more difficult as protocol schemes differ to a higher degree than function calls whose arguments and return values vary according to the API, but always adhere to the same ABI (Application Binary Interface) for a given platform.

A representative sample of the Xlib API is shown below.

```
Pixmap XCreatePixmap(Display *display, Drawable d, unsigned int width,
    unsigned int height, unsigned int depth)

int XFillRectangle(Display *display, Drawable d, GC gc, int x, int y,
    unsigned int width, unsigned int height)
```

XCreatePixmap is the sole resource acquisition call in the Xlib that creates a Pixmap,

XFreePixmap is sole resource destruction call, and XFillRectangle is one of a large number of operations that operate on a Drawable (either a Window or Pixmap). This is the general pattern for most resources, though some resources have multiple acquisition API calls and some have multiple destruction calls. Appendix A is a comprehensive list of the resource acquisition and destruction API calls.

All proposed solutions outlined depend on intercepting the Xlib API calls, parsing their arguments, and comparing utilized resource ids against resource ids extracted from intercepted acquire and destruction calls.

DSO Interposition

Calls to dynamic libraries such as Xlib can be intercepted by interposition (Curry, 1994) where a replacement library can be interposed between client and the normal dynamic library. The DSO Interposition technique can be used to implement an interposed library which overrides the functions found in libX11 that are of relevance to the case-study goals.

A DSO Interposition solution consists of:

1. For each resource creation/destruction API call collect a callstack within the interposed library.
2. Keep a map of the associated resources to those callstacks.
3. Forward the calls from replacement library to the real Xlib
4. Wrap the remainder of the Xlib API to test passed resource arguments against the maps of allocated and unallocated resources.
5. On detection of use of a deallocated or unallocated resource display an error including the call-stack of the detected location of the error and output and the code locations where they were created and destroyed when known.

6. On exit of the application output all callstacks of allocated resources which have had no matching deallocation call during the applications lifetime.

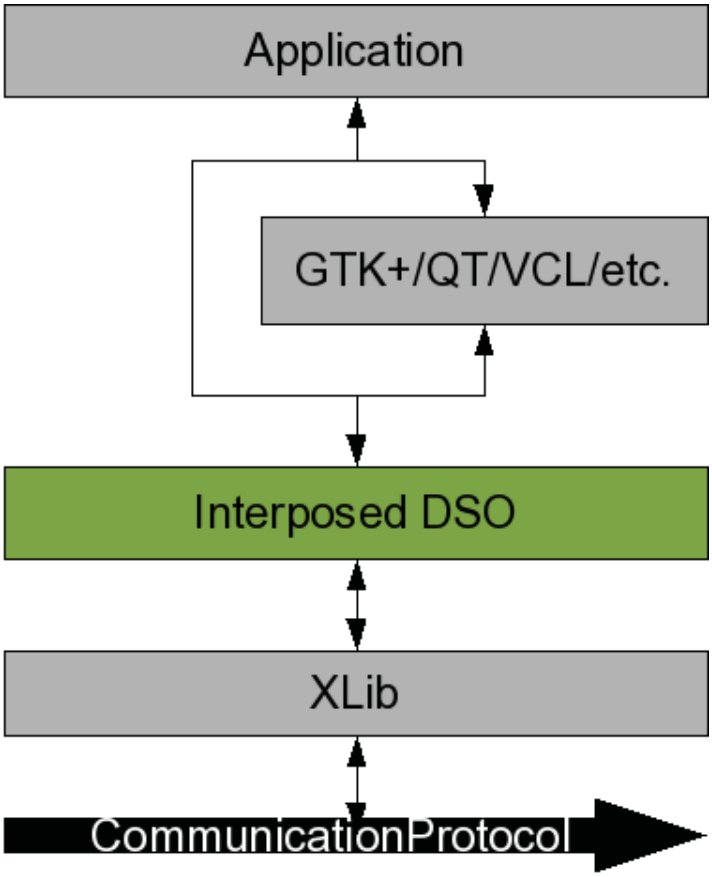

Illustration 3: Interposed Solution Stack

Given the representative sample of the API shown above, a sketch of the key components of the implementation is shown in Appendix B. This solution is relatively fast. The only substantial overhead is that caused by the execution of the replacement library functions. What this solution is capable of doing is:

1. detect and report the location of re-use of a released resources, report the location of the previous release of that resource, and report the location of the initial acquisition of the now released resource.

2. report resources acquired but never released

3. detect and report the location of use of an uninitialized resource

However, what this solution is incapable of doing is report the location where a utilized uninitialized resource was initially assigned its invalid value. By operating solely on a API interception level its impossible to detect the introduction of a value into the application that does not pass through the interposed library. The call-stack at the time of use of an uninitialized value may by happen-chance include the origin of the initial introduction of the invalid value, but in general the only mechanism capable of reporting the location of the origin of introduction of an uninitialized or invalidly initialized client-side integer handle is through some form of binary instrumentation origin tracking (Bond, et.al, 2007).

Dedicated Valgrind tool

The key of Valgrind in the context of the overall goal, on error detection, to report the location of introduction of an uninitialized resource handle to equal fidelity to that of reporting the location of deallocation and prior allocation of a now invalid handle, is Valgrind's origin tracking feature. Conveniently, the Valgrind framework provides mechanisms to generate and store callstacks on request and map them back to source code and line numbers. Valgrind also provides a DSO function wrapping mechanism. Both of these convenience mechanisms remove the necessity within the DSO Interposition technique to implement that additional infrastructure.

However running the entire client application through the Valgrind dynamic binary instrumentation framework is not without its costs. Running an application under Valgrind is at least 5 times slower than native execution, though this compares well to other similar dynamic binary interpreter frameworks such as Pin and DynamoRIO (Valgrind Developers, 2009).

A solution implemented using the Valgrind framework to create a dedicated tool to achive the case-study goals consists of:

1. Wrap the Xlib API within a new Valgrind tool.
2. For each resource creation/destruction API call request the Valgrind core to store a callstack.
3. Keep a map of the associated resources to those callstacks.
4. For the remainder of the Xlib API test passed resource arguments against the maps of allocated and unallocated resources.
5. On detection of use of a deallocated resource instruct the Valgrind core to display the callstack of the detection location and display the cached acquisition and destruction callstacks.
6. On detection of use of an unallocated or otherwise invalid resource instruct the Valgrind core to display the callstack of the detection location and request from Valgrind the origin tracking information from Valgrind for that integer value and display the origin location where that value was introduced.
7. On exit of the application output all callstacks of allocated resources which have had no matching deallocation call during the applications lifetime.

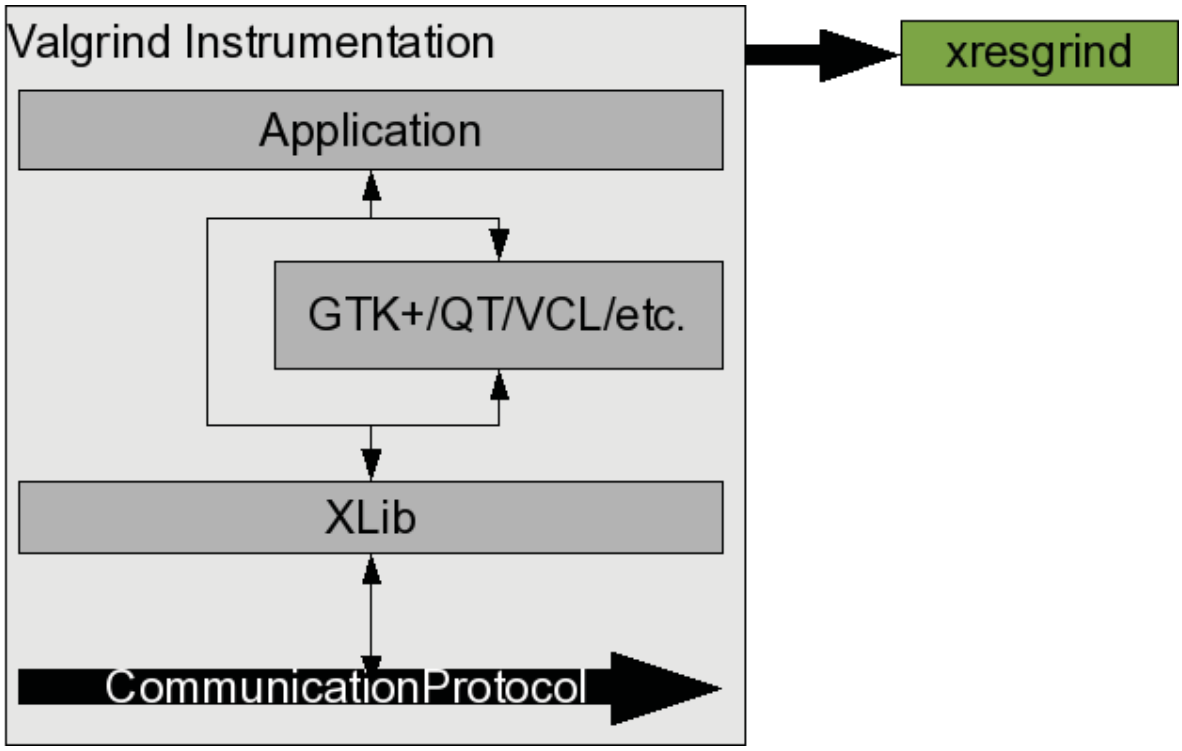

Illustration 4: Dedicated Valgrind Tool Stack

Given a representative sample of the API shown above, a sketch of the key components of the implementation is shown in Appendix C. This solution is capable of meeting the immediate goals, it can

1. detect and report the location of re-use of a released resources, report the location of the previous release of that resource, and report the location of the initial acquisition of the now released resource.

2. report resources acquired but never released and report the location of the acquisition

3. detect and report the location of use of an uninitialized resource, **and** report the location or the origin of that invalid value.

Using the Valgrind dynamic binary instrumentation framework gains the ability to detect origin information for uninitialized values, at the cost of increased runtime over, but retaining the ability to execute on unmodified binaries. However, such a custom tool is hard-coded to the case-

study and is not amenable to easy extension to the wider case. To extend the tool to handle different APIs or other classes of similar problems it must be manually extended.

Final Hybrid Architecture

An optimal solution that supports origin tracking is a Valgrind-based tool which offers an extensible route to easily handle similar problems. An open design decision is whether to offer a tool which can just be used to check for out-of-process resource misuse, albeit one that is extensible to multiple situations, or to extend the existing Valgrind memory checking tool Memcheck.

Memcheck tests for analogous in-process memory allocations/deallocations, detects use of unallocated memory and deallocated memory and reports on memory leaks. Memcheck has also been extended to track file descriptors and report on double closes of file descriptors, use of closed and unopened file descriptors and report on file descriptors that are never closed. Extending Memcheck to support reporting of arbitrary resource leaks and misuse via an extensible interface has the advantage of providing a tool which is capable of reporting multiple classes of both in-process and out-of-process errors at the same time, giving the end programmer a single, simpler mechanism to test for resource leaks and misuse regardless of the type of resource.

The final architecture which supports these desirable features consists of modifications to Memcheck to support tracking and reporting on arbitrary resources

1. A Memcheck API which can be used by code executed **inside** the Valgrind runtime dynamic binary instrumentation.
2. A set of interposed DSOs each of which implements a wrapper around the underlying libraries which provide and consume the resource handles and communicate when executed under Valgrind with the extended Memcheck tool using the Memcheck API to inform Memcheck of the resources created, destroyed and request validation of each resource use.

The in-process resource checks on memory and file descriptors of Memcheck are unaffected by these extensions.

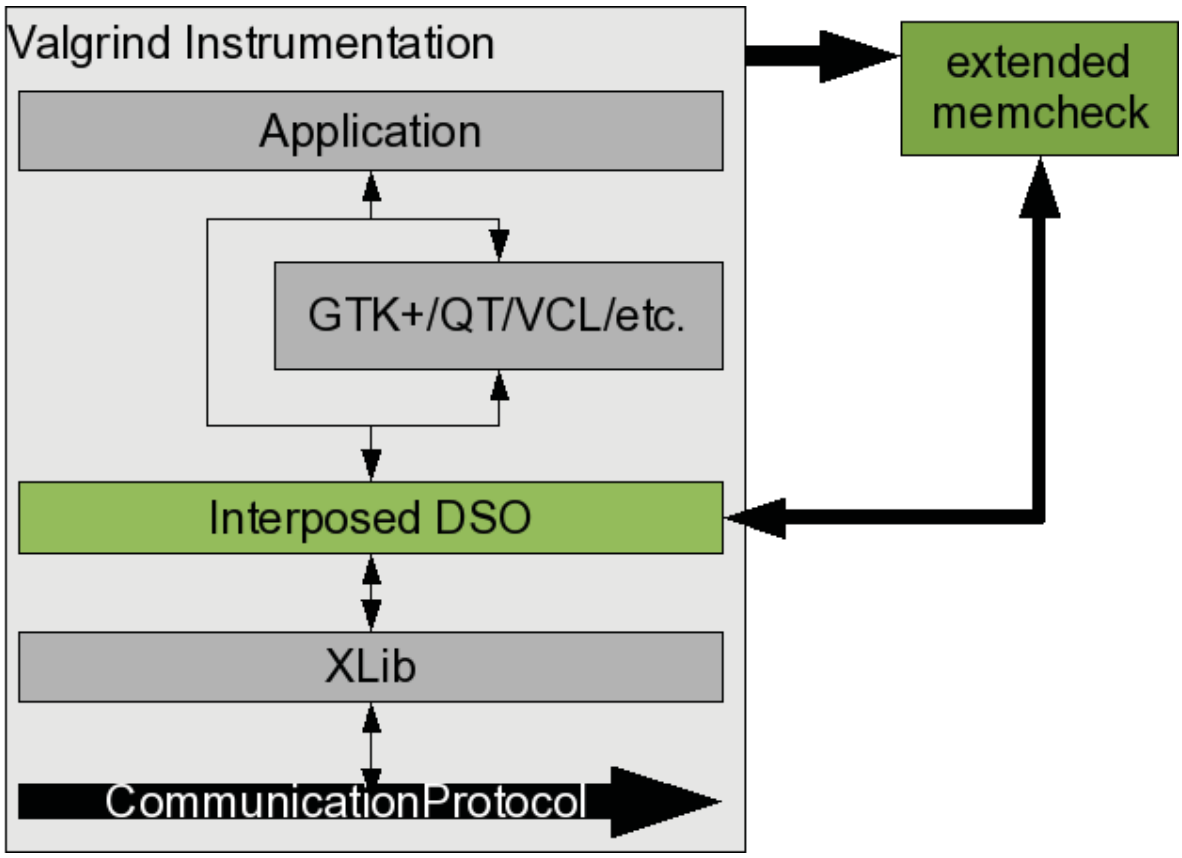

Illustration 5: Final Valgrind Stack

The architectural diagram illustrates the basic concepts of the design. The target application runs under the extended Memcheck Valgrind tool, and Memcheck makes its normal tests on all memory and file descriptor uses. Calls to the various APIs are intercepted by the interposed DSOs which use runtime Memcheck hooks to inform Memcheck of the client's use of integer handles to out-of-process resources and categorize them as acquire, release or use operations.

So informed, the extended Memcheck can then report on out-of-process resources equivalently to in-process ones. The use of an interposed DSO to inform Memcheck of resource events enables a relatively easy extension mechanism to simply create additional wrapper DSOs for similar situations to the case-study which can reuse the generic Memcheck logic.

API

Valgrind supports an API which can be used by code executed inside the Valgrind runtime dynamic binary instrumentation, i.e. VALGRIND_DO_CLIENT_REQUEST which an application running under a given Valgrind tool can use to communicate with the controlling Valgrind instance, e.g.

```
VALGRIND_DO_CLIENT_REQUEST(..., VG_USERREQ__MAKE_MEM_UNDEFINED, _qzz_addr,
```

to inform Memcheck that a given range of memory should be considered undefined. Using this API as a basis, support for informing Memcheck of remote resources can be added though the creation of an API of:

VG_USERREQ__ACQUIRE_RESOURCE(handle, type, parent)

VG_USERREQ__USE_RESOURCE(handle, type)

VG_USERREQ__RELEASE_RESOURCE(handle, type)

Where a type is a simple integer id to disambiguate resources with potentially the same id but of different types, e.g. Font versus Pixmap.

The semantics are that the destruction of a parent id as passed to extended Memcheck with VG_USERREQ__ACQUIRE_RESOURCE in the creation on a child implies the automatic destruction of all children of that parent id. Pixmaps which require the existence of another Drawable in their creation API by this rule do not inform Memcheck that the reference Drawable is a parent, but instead pass a NULL id as a parentless resource. Adding some complexity to this area is that the Xlib API provides a XDestroySubwindows() which destroys the children of a window, but not the window itself. To support this concept, the Memcheck API additionally requires a

VG_USERREQ__RELEASE_SUBRESOURCES(handle, type) API call.

Implementation

Many Xlib API calls take Drawables as arguments, in order to support checking the validity of arguments passed to such methods both Window and Pixmaps must be considered. The simplest solution is to specify that the type argument in VG_USERREQ__USE_RESOURCE is a bitfield where types can be ORed together to indicate that any of the types specified is legal for use.

Unlike the architecture proposed for a dedicated Valgrind tool, these APIs are primarily for use within an interposed DSO to provide it with a way to communicate with the extended Memcheck Valgrind tool when the application it is linked to is executing under Valgrind. The extended Memcheck has the task of tracking the resource usage, while the DSO has the task of feeding the events into Memcheck. A representative section of the DSO side of the implementation to handle a subset of the API that demonstrates acquiring a resource, releasing a resource, and capturing the use of a resource which may be of multiple types is shown in Appendix D.

The crucial components on the extended Memcheck are the handlers for these events, and the built-in Memcheck support for origin tracking and stacktrace recording. The handlers for the events in this hybrid model follows the same pattern as shown for the dedicated tool in Appendix C, while a simplified demonstration of the use of origin tracking to report the location where an uninitialized value entered the program flow is shown in Appendix E to illustrate the key issues for the VG_USERREQ__USE_RESOURCE handler.

The corresponding handlers for the acquire and release operations store the stack traces of these events and associated them with the handle id. The release handler also makes use of the use handler in order to report on double releases, or releases of acquired resources.

The Memcheck-side is completely unaware of and independent of the Xlib API. It deals solely in terms of integers being flagged by the external helper DSO as handles to remote resources. The helper DSO provides the hierarchical information for each handle in order for Memcheck to

infer that acquired child handles have become invalid.

A feature of the X Windowing System that adds a certain degree of complexity to the specific case study is that all Windows must be the child of some other window. A clients toplevel window therefore must itself be the child of some other window, i.e. the root window of the hierarchy is not created by the client, but instead exists before the instantiation of the client but the client may, and in practice already certainly must, operate on its resource handle. A number of other instances of this situation where a client makes use of resource ids not created by itself, either as parents for windows or for querying for shared desktop resources, exist.

In order to not trigger false positives of resource misuse reports on use of resources before allocation, these resource handles must be explicitly excluded by the interposed DSO from the use events reported to Memcheck. Examples include operations on the SelectionOwner window id in order to paste content from one application to another, or creation of a toplevel window as a child of the DefaultRootWindow id. In these two examples non-destructive operations on those window ids should be filtered out before reporting to Memcheck to suppress false positives, while attempts to destroy those windows can be allowed to pass through to Memcheck to be reported as invalid operations.

The generic Memcheck extension supports sufficient operations to capture the relations between resources that are necessary to maintain a model of the validity and life-cycle of remote X Windowing resources in order to report meaningful information about their misuse. But it is insulated from the details and semantics of that particular case study API. It is unaware of the actual API in use and sufficiently flexible to be reused without modification by alternative interposition DSOs which implement different remote resource APIs that follow a similar acquire/release pattern.

This architecture enables individual quirks of the API to be handled within each supplementary DSO, such as the Xlib SelectionOwner window issue, without compromising the relative simplicity of the general purpose reusable core.

Testing

Test Harness

To create a comprehensive test-suite to prove that the tooling is sufficient to handle all detectable scenarios, the Xlib API must be analyzed to determine the number of different possible remote resources, the calls to acquire them, the calls to release them and aspects of the semantics of use.

Manual inspection of the API and associated documentation (Gettys & Scheifler, 2002) shows six possible server-side resources, Colormaps, Cursors, Fonts, Pixmaps, Windows and Graphic Contexts. Of these six possibilities, inspection of Graphic Contexts shows that the client-side integer handle for the server-side Graphic Contexts can only be acquired or released as part of a local structure which is dynamically allocated or deallocated by the Xlib internals. As such, using the final architecture of an extended Memcheck solution, misuse of this category of remote resource will be already automatically detected through the standard Memcheck local memory tests. Leaks, use before acquisition and use after release, will all be captured by the standard local tests, so this category of resource can be discarded from consideration.

The Xlib API can then be categorized into calls that return or take a handle to a remote resource versus those that don't. In this case the second category is clearly not relevant and can be discarded. The first category can be further subdivided into acquire calls, release calls, subrelease calls and usage calls.

The API calls that return or take a remote resource can be further categorized into *primitive* and *utility* calls. Utility calls are those calls which do not directly operate on the resource, but pass the resource through other intermediate API calls. The remaining API calls, which do not decompose into other APIS calls, are our primitives. These comprise the the subset of calls that are

necessary to intercept in the interposed DSO, the utility calls can be discarded from consideration as their use of remote resources is delegated to the primitives. As a concrete example the XCreateFontCursor API call is a utility function which wraps XCreateGlyphCursor and so does not need to be explicitly intercepted.

The test-harness then consists of tests that exercise capture of every primitive entry point to acquire and release each remaining category of resource. Each category of resource requires tests to validate:

1. No errors: That legal use of the resource does not trigger a false positive
2. Leak: That omission of a release call is detected
3. Use before acquire: That use of an never-valid handle is detected
4. Use after release: That use of a released handle is detected
5. Double Free: That an attempt to release an already released handle is detected.

Strictly speaking double-frees are a sub category of “Use after release”. But is an important grouping worthy of explicit test-cases as historically the local memory equivalent of duplicate use of the “free” memory release call has been a source of many security exploits (MITRE, 2009).

The source of the resulting test cases derived from the above analysis and the corresponding output from the modified Memcheck can be found in Appendix A. For the purposes of the test-harness, all acquire and release paths are exercised, but due to the space required only a representative selection of the API that operates on each category of resource is selected.

Field Testing

The project goal is that the tooling is sufficient to detect resource misuse in unmodified real-world large application binaries. A reasonable target is OpenOffice.org, a large office suite available for Linux and other UNIX operating systems which has been shown to suffer from X

resource leaks in the past and suspected of resource leaks at present.

Detecting Known Issues

McCullagh (2008) opened a bug against OpenOffice.org 2.3 to report "both Impress and Writer crashing thin clients where a large amount of image data is placed in the document. The application pushes the pixmap image data across onto the X server which is forced to allocate memory to store it". Traditional debugging discovered that the root cause was a remote X Pixmap leak where matching a XFreePixmap call on the result of XCreatePixmap was missing and subsequently solved for OpenOffice.org 2.4. The difficulty in manually discovering the origin of this leak was a prime motivation for this thesis.

This bug is known to be a Pixmap leak and the location of the leak also known, so it provides a real-world scenario in a large application which the tooling should theoretically be able to discover and report accurately on. The error report can be compared against the known location of the error to validate the results.

For the purposes of the experiment the bugfix was reverted from a local copy of OpenOffice.org 3.1 to recreate the leak and the binary started under the modified Memcheck tooling and immediately quit after start up was completed. OpenOffice.org successfully executed under the framework, and at normal program termination the following trace was output by the tooling:

```
==13354== Resource 0x46002fb of class 2 never released, acquired at
==13354==    at 0x400E2A6: XCreatePixmap (xr_intercepts.c:255)
==13354==    by 0x548154F: SalGraphics::DrawAlphaBitmap(SalTwoRect const&,
   SalBitmap const&, SalBitmap const&, OutputDevice const*)
   (salgdilayout.cxx:793)
==13354==    by 0x541322D: OutputDevice::ImplDrawAlpha(Bitmap const&,
  AlphaMask const&, Point const&, Size const&, Point const&, Size const&)
  (outdev2.cxx:1983)
==13354==    by 0x5413CF8: OutputDevice::ImplDrawBitmapEx(Point const&, Size
   const&, Point const&, Size const&, BitmapEx const&, unsigned long)
   (outdev2.cxx:891)
==13354==    by 0x54141FA: OutputDevice::DrawBitmapEx(Point const&, Size const&,

==13354==    by 0x53E1C6E: ImplImageBmp::Draw(unsigned short, OutputDevice*,
   Point const&, unsigned short, Size const*) (impimage.cxx:550)
==13354==    by 0x54145B7: OutputDevice::DrawImage(Point const&, Image const&,
```

```
  unsigned short) (outdev2.cxx:1204)
```

The reported source-code location of “salgdilayout.cxx:793” identified the following line of code

```
 Pixmap aAlphaPM = XCreatePixmap( pXDisplay, hDrawable_, rTR.mnDestWidth,
   rTR.mnDestHeight, 8 );
```

which correctly identifies the source of the known bug where the handle returned by this XCreatePixmap was never destroyed with a matching XFreePixmap, accumulating X Server resources leading to an eventual resource starvation.

Detecting Unknown Issues

The modifications were shown to be capable of discovering the known issue, but another issue was also reported on exit from a basic start-up and exit cycle, i.e.:

```
==29004== Resource 0x4800015 of class 8 never released, acquired at
==29004==    at 0x400E4CE: XCreatePixmapCursor (xr_intercepts.c:292)
==29004==    by 0xC7A2196: x11::SelectionManager::createCursor(char const*,
  char const*, int, int, int, int) (X11_selection.cxx:277)
==29004==    by 0xC7A4624: x11::SelectionManager::initialize(
  com::sun::star::uno::Sequence<com::sun::star::uno::Any> const&)
 (X11_selection.cxx:443)
==29004==    by 0xC7A4AD5: x11::SelectionManagerHolder::initialize(
  com::sun::star::uno::Sequence<com::sun::star::uno::Any> const&)
```

Some investigation of this report showed that the detection of an additional leak is accurate. The Cursors created through XCreatePixmapCursor at the reported location had no matching XFreeCursor call creating a very small previously unknown leak. Due to this discovery a fix was created and submitted for inclusion into OpenOffice.org 3.2 (McNamara, 2009).

The known bug reported by McCullagh is not the only reported bug where xrestop indicated the likely presence of an X resource leak. Erickson (2009) also reported a bug discovered in OpenOffice.org 2.4.1 where “Working in OpenOffice Impress/Presentation is fine until starting "Slide Show", which caches excessive amounts of pixmap data to thin-client memory. This is bad because thin clients generally have very little amount of RAM (64-128MB is typical), and when all of the RAM is exhausted by something like excessive pixmap memory usage, the session simply

crashes outright, causing data loss.”

The symptoms reported are similar to the bug reported by McCullagh but were unchanged by the fix for that problem. The root problem was unknown, but xrestop showed an increase of two Pixmaps and a Window after every new use of the slide show making this example a candidate for investigation with this tooling. Running the presentation software under the tooling and starting and exiting the Slide Show provided a vast stack trace, an abbreviated version of which appears here that contains the major stack frames, that identifies the locations where an X resource related to starting a slide show has leaked.

```
==29471== Resource 0x4400c14 of class 2 never released, acquired at
==29471==    at 0x400E2A6: XCreatePixmap (xr_intercepts.c:255)
==29471==    by 0x62C468F: gdk_pixmap_new (in
  /usr/lib/libgdk-x11-2.0.so.0.1700.0)
==29471==    by 0x54F4588: Window::ImplInit(Window*, long long,
  SystemParentData*) (window.cxx:824)
==29471==    by 0x548A789: ImplBorderWindow::ImplInit(Window*, long long,
  unsigned short, SystemParentData*) (brdwin.cxx:1887)
==29471==    by 0x548A89A: ImplBorderWindow::ImplBorderWindow(Window*,
  SystemParentData*, long long, unsigned short) (brdwin.cxx:1922)
==29471==    by 0x5500D79: WorkWindow::ImplInit(Window*, long long,
  SystemParentData*) (wrkwin.cxx:76)
==29471==    by 0x55010B3: WorkWindow::WorkWindow(Window*, long long)
  (wrkwin.cxx:124)
==29471==    by 0xD6BAA37: sd::SlideShow::StartFullscreenPresentation()
  (slideshow.cxx:1204)
==29471==    by 0xD6BABF1: sd::SlideShow::startWithArguments(
  com::sun::star::uno::Sequence<com::sun::star::beans::PropertyValue> const&)
```

Examining the reported stack to discover the owner of the acquired Pixmaps reported to be leaked showed that the object (WorkWindow in the stacktrace above) referencing the remote resources was not itself correctly released. Once discovered the a simple fix for the leak was submitted for consideration for inclusion in OpenOffice.org 3.2 (McNamara, 2009).

These results show that the tooling is practical to use with a large real-world application, discovers and reports accurately on X resource misuse, dramatically reducing the effort required to identify the existence and source-code location of the introduction of those errors.

Conclusions

The presented solution operates on unmodified binaries and reports locations where a resource was acquired but not released and locations where invalid resource handles are used. On use of an invalid resource handle, the location where the resource was either previously deallocated or where the uninitialized handle was introduced into the program flow are shown. There are vital pieces of information to guide the programmer in solving the detected flaws.

The tooling is not tied to one specific category of out-of-process resource tracking and can be extended to support any similar situation where client-side code manages out-of-process resources through an API which can be intercepted. Multiple DSOs to intercept different APIs can coexist at runtime communicating with the central hub to support checking multiple APIs at the same time.

What has been demonstrated by this thesis is a practical architecture and set of techniques to enable building debugging tools that are aware of out-of-process resources which otherwise can not be seen by current in-process resource monitoring debugging tools such as traditional bounds checkers and memory checkers. The architecture has been shown to be capable of successfully automatically discovering and correctly reporting errors on misuse of remote resources in the X Windowing System case-study to an equivalent degree of quality as performed by a standard Valgrind Memcheck tool for in-process memory errors.

Future Work

There is scope for future work in both the specific X Resource tracking tool plugin and the wider resource tracking architecture.

Image Grabs on Drawables: Specific to the X Window case the tooling could be improved by adding features to the API interceptor to capture image grabs of Drawables at destruction time to provide a view of them in a debugging GUI to help visually identify what was last referenced by a handle if it is later used after becoming invalid. Similarly at exit time the tooling could be extended to take image grabs of the contents of leaked Drawables.

The core generic part of the tool could be enhanced to help isolate difficult to debug problems that are not specific to the X Windowing System.

Instrument a particular execution path: A trigger mechanism to control where API interception begins and ends during a clients lifecycle would enable verifying that a given execution path's resource utilization matches expectations. A particular execution path might not leak resources from the perspective that all resources are eventually released, but it may be considered to *logically* leak (Maebe, 2004) where resources should have been released earlier than they eventually are. Support for resource checking between check points would enable detection of such logical leaks.

Time stamping: Enable recording time-stamp information for operations on resource handles and enable supporting arbitrary queries to search for long lived resources that are unused for long periods of time prior to eventual destruction. Getting access to this information would enable discovery of potential lost resource optimization opportunities.

Support Reparenting Resources: Some API calls may *reparent* a resource where a given resource is moved from one part of the hierarchy to another. There is currently no support for this

feature, so there is the theoretical possibility of a resource being reported as leaking when it has in fact be reparented under another resource which was subsequently destroyed automatically destroying its subresources.

Implement More API Interceptors: The case-study implemented one interposed DSO for one API, creating extra plugins for other APIs that control remote resource through handles (e.g. APIs that control remote database resources) would exercise the core to identify if the supported semantics of acquire, release, and sub-release are sufficient for the general case or if further extensions to the internal API is required to support additional concept used by other APIs, e.g. speculatively a given API might include a call to destroy an entire category of resources, a concept which the current core doesn't support.

Formal API description language: Manually examining the Xlib API documentation to determine whether a call that returns a resource places a responsibility to release ownership of that resource to the caller or not, and if so, what is the correct release function was fraught with difficulty, e.g the documentation for. XCreateFontCursor makes no mention if the returned Cursor should be released by a client and no mention of a corresponding release function, but a XFreeCursor call is separately documented and references XCreateFontCursor. Wrapping the APIs by manually writing wrapping functions with the same signatures as the API that forward to the true API is a tedious task that should be possible to mostly automate.

A formal API description language with support for indicating which out parameters are the responsibility of the caller to release and with what matching API would resolve these ambiguities. Such a language should have notation for indicating if out parameters are part of a hierarchical model where children are automatically destroyed on destruction of a parent and mechanisms for describing how an API call may modify the resource hierarchy, e.g. API calls that remove a child from one hierarchical tree and add it to another. A language along these lines would enable automated processing of APIs to generate interposition DSOs without tedious manual parsing of

documentation and provide a common language that further tools that perform dynamic and static analysis of software could reuse.

Concluding Remarks

This thesis has presented a practical architecture for tracking out-of-process resources residing in a server, but controlled by a client, in order to automatically at client-side report on resource leaks and other misuses by an individual client. The techniques shown can be applied to the general case of improving the quality of Client/Server Architectures where equivalent defects can otherwise go undetected, for example:

Remote Procedure Calls: Remote Procedure Call (RPC) technology enables clients to execute procedures in another address space, typically on a remote server. Server-side resources created or controlled over RPC by a client are vulnerable to the same defects as described in the case-study. The techniques shown here can be used to extract acquire/release ownership rules of resources controlled by a given RPC API and married to the demonstrated model in order to similarly detect volition of ownership rules of remote resources. APIs based on conceptually similar out-of-process middleware technology such as Common Object Model or CORBA are equally vulnerable to clients accidentally either exhausting server-side resources, or exhausting the maximum available allocation for a single client. Applications based on these remote invocation technologies can benefit from debugging tools that automatically track their remote resource utilization and report client-side locations of remote resource leaks and misuse.

Remote Database Connectivity: The Open Database Connectivity (ODBC) API is a standard that allows a client to access remote databases. Basic errors in client-side ODBC applications are capable of creating effective leaks at server side where a client request causes memory to be allocated in the server but the client omits the call that directs the server to release the memory e.g. “only the SQL_DROP option of the SQLFreeStmt API actually frees all memory

associated with the handle. SQL_CLOSE and SQL_UNBIND do not ... each statement handle allocated by the application also results in memory allocated on the server" (IBM, 2008). Similar possibilities exist in other mechanisms for accessing remote databases, e.g. Java Database Connectivity and ActiveX Data Object.

Wrapping the remote database APIs to record which calls return handles to acquired remote resources that need to be explicitly released with specific calls would enable the client-side programmer to detect and debug these errors.

Works Cited

Allum, M. (2003). XResTop is A 'top' like tool for monitoring X Client server resource usage. Retrieved Jul 16, 2009 from http://www.freedesktop.org/wiki/Software/xrestop

Allum, M. (2008). Xres. Retrieved Jul 16, 2009 from http://www.x.org/wiki/PeopleProjectsPresentation

Bond, M.D., Nethercote, N., Kent, S.W., Guyer, S.Z., McKinley, K.S. (2007). Tracking bad apples: reporting the origin of null and undefined value errors. Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. ACM.

Cantrill, B.M., Shapiro, M.W., Leventhal, H.L. (2004). Dynamic Instrumentation of Production Systems. Proceedings of the USENIX 2004 Technical Program. USENIX. Retrieved Feb 17, 2009 from http://www.usenix.org/event/usenix04/tech/general/full_papers/cantrill/cantrill_html/

Curry, T.W. (1994). Profiling and Tracing Dynamic Library Usage via Interposition. Proceedings of the USENIX Summer 1994 Technical Conference. USENIX. Retrieved Feb 17, 2009 from http://www.usenix.org/publications/library/proceedings/bos94/full_papers/curry.ps

Drepper, U. (2006). How To Write Shared Libraries. Retrieved Jul 16, 2009 from http://people.redhat.com/drepper/dsohowto.pdf

Dumitran, D. (2007). Fixing File Descriptor Leaks. Master's thesis, Massachusetts Institute of Technology. Retrieved Feb 17, 2009 from http://dspace.mit.edu/handle/1721.1/41645

Erickson, J. (2009). Starting "Slide Show" caches pixmap data excessively, crashing thin clients. Retrieved Jul 10, 2009 from http://qa.openoffice.org/issues/show_bug.cgi?id=97906

Gettys, J., Scheifler, R.W. (2002). Xlib − C Language X Interface. X Consortium. Retrieved Feb 18, 2009 from http://www.xfree86.org/current/xlib.pdf

Giraldeau, F., Dault, J.M, des Ligneris, B. (2006, September). MILLE-XTERM and LTSP. Linux Journal. Specialized Systems Consultants, Inc. Seattle, WA.

Hastings, R., Joyce, B. (1992). Purify: Fast detection of memory leaks and access errors. Proceedings of the Winter USENIX Conference. USENIX. Retrieved Feb 17, 2009 from http://opera.cs.uiuc.edu/probe/reference/debug/dynamic/purify_92.pdf

Herrb, M., Hopf. M. (2005). New Evolutions in the X Window System. Proceedings of EuroBSDCon 2005.

IBM (2008). ODBC S1001 / HY001 Memory Allocation Failure. Retrieved Aug 18, 2009 from http://www-01.ibm.com/support/docview.wss?uid=nas12baf94c1a1ddbef48625660a0075db8a

Lee, K. (1992). Graphics Effects by Manipulating X Colormaps. The X Journal, May 1992.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J, Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. ACM.

Maebe, J., Ronsse, M., De Bosschere , K. (2004). Precise detection of memory leaks. Second International Workshop on Dynamic Analysis. IEEE.

Manrique, D. (2001). X Window System Architecture Overview HOWTO. Retrieved Jul 16, 2009 from http://www.faqs.org/docs/Linux-HOWTO/XWindow-Overview-HOWTO.html

McCullagh, G. (2008) OOo caches large pixmaps to X server, crashing the X server. Retrieved Aug

18, 2008 from http://www.openoffice.org/issues/show_bug.cgi?id=85321

McNamara, C. (2007). backtraces and prelink. Retrieved Jul 16, 2009 from http://blogs.linux.ie/caolan/2007/04/16/backtraces-and-prelink/

McNamara, C. (2009). dtrans: xresource leaks. Retrieved Jul 29, 2009 from http://qa.openoffice.org/issues/show_bug.cgi?id=102133

McNamara, C. (2009). sd: BasicViewFactory leaks windows. Retrieved Jul 10, 2009 from http://qa.openoffice.org/issues/show_bug.cgi?id=102142

MITRE. (2009). Common Weakness Enumeration 415: Double Free. Retrieved Jul 27, 2009 from http://cwe.mitre.org/data/definitions/415.html

Nethercote, N., Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. ACM.

Nethercote, N., Seward, J. (2007). How to shadow every byte of memory used by a program. Proceedings of the 3rd international conference on Virtual execution environments. ACM.

Nethercote, N., Walsh, R., Fitzhardinge, J. (2006). Building Workload Characterization Tools with Valgrind. IEEE International Symposium on Workload Characterization. IEEE.

Rayside, D. Mendel, L. (2007). Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks . Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM.

Scheifler, R.W., Gettys, J. (1986). The X Window system. Transactions on Graphics (TOG), 5(2). ACM.

Srivastava, A., Eustace A. (2004). ATOM: a system for building customized program analysis tools.

SIGPLAN Notices, 39(4). ACM.

Sun Microsystems (2009). BigAdmin System Administration Portal - DTrace. Retrieved Jul 16, 2009 from http://www.sun.com/bigadmin/content/dtrace/

Valgrind Developers (2009). Valgrind Documentation, Release 3.4.0. Retrieved Aug 18, 2009 from http://valgrind.org/docs/manual/valgrind_manual.pdf

Appendix A: Test Matrix Results

	Colormap	Cursor	Font	Pixmap	Window
Leak	Detected 0 errors Detected 1 leak	Detected 0 errors Detected 1 leak	Detected 0 errors Detected 1 leak	Detected 0 errors Detected 1 leak	Detected 0 errors Detected 1 leak
Double Release	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks
Use after release	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks
Use before acquire	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks	Detected 1 error Detected 0 leaks
No Errors	Detected 0 errors Detected 0 leaks	Detected 0 errors Detected 0 leaks	Detected 0 errors Detected 0 leaks	Detected 0 errors Detected 0 leaks	Detected 0 errors Detected 0 leaks

Colormap: Leak

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    Colormap map = XcreateColormap(dpy, /*Never released*/
        DefaultRootWindow(dpy),
        DefaultVisual(dpy, DefaultScreen(dpy)), AllocNone);

    /*Does not release map, and return-map not released*/
    XCopyColormapAndFree(dpy, map);

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 4 never released, acquired at
   at 0x400F3F6: XCreateColormap (xr_intercepts.c:484)
   by 0x80485C5: main (testLeakColormap.c:6)

Resource 0x4c00002 of class 4 never released, acquired at
   at 0x400F4A8: XCopyColormapAndFree (xr_intercepts.c:498)
   by 0x80485DD: main (testLeakColormap.c:11)

RESOURCE LEAK SUMMARY:
   definitely lost: 2 resources.
```

Cursor: Leak

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);

    XColor bcolor;
```

```
    Pixmap p;
    Cursor c;

    p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 10, 10, 1);
    c = XCreatePixmapCursor(dpy, p, None, &acolor, &bcolor, 0, 0);

    XFreePixmap(dpy, p); /*Pixmap released, but cursor isn't and leaks*/

    XCloseDisplay(dpy);

}
```

```
Resource 0x4c00002 of class 8 never released, acquired at
   at 0x400EBA5: XCreatePixmapCursor (xr_intercepts.c:356)
   by 0x804861A: main (testLeakCursor.c:12)

RESOURCE LEAK SUMMARY:
   definitely lost: 1 resources.
```

Font: Leak

```
#include <X11/Xlib.h>

int main(void)
{
    Font f;
    Display *dpy = XOpenDisplay(NULL);

    f = XLoadFont(dpy, "fixed"); /*font never released*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 10 never released, acquired at
   at 0x400EFE3: XLoadFont (xr_intercepts.c:416)
   by 0x8048520: main (testLeakFont.c:8)

RESOURCE LEAK SUMMARY:
   definitely lost: 1 resources.
```

Pixmap: Leak

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);

    XCreatePixmap(dpy, DefaultRootWindow(dpy), 200, 100, 1); /*never released*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 2 never released, acquired at
   at 0x400E906: XCreatePixmap (xr_intercepts.c:317)
   by 0x804856A: main (testLeakPixmap.c:7)

RESOURCE LEAK SUMMARY:
   definitely lost: 1 resources.
```

Window: Leak

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);

    Window w = XCreateSimpleWindow(dpy, /*never released*/
        DefaultRootWindow(dpy),
        0, 0, 200, 100, 0,
        BlackPixel(dpy, DefaultScreen(dpy)),
        WhitePixel(dpy, DefaultScreen(dpy)));

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 1 never released, acquired at
   at 0x400DEDE: XCreateSimpleWindow (xr_intercepts.c:174)
   by 0x8048610: main (testLeakWindow.c:7)

RESOURCE LEAK SUMMARY:
   definitely lost: 1 resources.
```

Colormap: Double-release

```
#include <X11/Xlib.h>

int main(void)
{
    XColor c;
    Display *dpy = XOpenDisplay(NULL);
    Colormap map = XCreateColormap(dpy, DefaultRootWindow(dpy),
        DefaultVisual(dpy, DefaultScreen(dpy)), AllocNone);

    XAllocColor(dpy, map, &c);

    XFreeColormap(dpy, map);
    XFreeColormap(dpy, map); /* map already released!*/

    XCloseDisplay(dpy);

}
```

```
Resource 0x4c00001 of class 4 used, but is already released
   at 0x4031542: XFreeColormap (xr_intercepts.c:510)
   by 0x804863D: main (testDoubleFreeColormap.c:13)
Release was at
   at 0x4031542: XFreeColormap (xr_intercepts.c:510)
   by 0x8048629: main (testDoubleFreeColormap.c:12)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Cursor: Double-release

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    XColor acolor;
    XColor bcolor;
    Pixmap p;
    Cursor c;

    p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 10, 10, 1);
    c = XCreatePixmapCursor(dpy, p, None, &acolor, &bcolor, 0, 0);

    XDefineCursor(dpy, DefaultRootWindow(dpy), c);

    XFreeCursor(dpy, c);
    XFreeCursor(dpy, c); /*c already released*/
    XFreePixmap(dpy, p);

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00002 of class 8 used, but is already released
   at 0x400EDE1: XFreeCursor (xr_intercepts.c:389)
   by 0x80486F4: main (testDoubleFreeCursor.c:17)
Release was at
   at 0x400EDE1: XFreeCursor (xr_intercepts.c:389)
   by 0x80486E0: main (testDoubleFreeCursor.c:16)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Font: Double-release

```
#include <X11/Xlib.h>

int main(void)
{
    XFontStruct *fontinfo;
    Font f;
    Display *dpy = XOpenDisplay(NULL);

    f = XLoadFont(dpy, "fixed");
```

```
    if (fontinfo)
    {
        /*XFreeFont automatically calls CloseFont on the fontid*/
        XFreeFont(dpy, fontinfo);
    }
    /*so calling UnloadFont on f is a double-free*/
    XUnloadFont(dpy, f); /* f already released */

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4a00001 of class 10 used, but is already released
   at 0x400F13B: XUnloadFont (xr_intercepts.c:444)
   by 0x804860B: main (testDoubleFreeFont.c:17)
Release was at
   at 0x400F1F3: XFreeFont (xr_intercepts.c:457)
   by 0x80485F7: main (testDoubleFreeFont.c:14)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Pixmap: Double-release

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);

    Pixmap p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 200, 100, 1);

    XFreePixmap(dpy, p);
    XFreePixmap(dpy, p); /* p is released!*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 2 used, but is already released
   at 0x400EA07: XFreePixmap (xr_intercepts.c:333)
   by 0x80485C6: main (testDoubleFreePixmap.c:10)
Release was at
   at 0x400EA07: XFreePixmap (xr_intercepts.c:333)
   by 0x80485B2: main (testDoubleFreePixmap.c:9)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Window: Double-release

```
#include <X11/Xlib.h>

int main(void)
```

```
    Display *dpy = XOpenDisplay(NULL);

    Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
        0, 0, 200, 100, 0,
        BlackPixel(dpy, DefaultScreen(dpy)),
        WhitePixel(dpy, DefaultScreen(dpy)));

    XSelectInput(dpy, w, ButtonPressMask | KeyPressMask);
    XMapWindow(dpy, w);

    XDestroyWindow(dpy, w);
    XDestroyWindow(dpy, w); /*w is released!*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 1 used, but is already released
   at 0x400E164: XDestroyWindow (xr_intercepts.c:209)
   by 0x80486CC: main (testDoubleFreeWindow.c:16)
Release was at
   at 0x400E164: XDestroyWindow (xr_intercepts.c:209)
   by 0x80486B8: main (testDoubleFreeWindow.c:15)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Colormap: Use after release

```
#include <X11/Xlib.h>

int main(void)
{
    XColor c;
    Display *dpy = XOpenDisplay(NULL);
    Colormap map = XCreateColormap(dpy, DefaultRootWindow(dpy),
        DefaultVisual(dpy, DefaultScreen(dpy)), AllocNone);

    XFreeColormap(dpy, map);
    XAllocColor(dpy, map, &c); /*map is released!*/

    XCloseDisplay(dpy);

}
```

```
Resource 0x4a00001 of class 4 used, but is already released
   at 0x400F5F3: XAllocColor (xr_intercepts.c:521)
   by 0x8048629: main (testDeInitializedColormap.c:11)
Release was at
   at 0x400F542: XFreeColormap (xr_intercepts.c:510)
   by 0x804860D: main (testDeInitializedColormap.c:10)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 22 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Cursor: Use after release

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    XColor acolor;
    XColor bcolor;
    Pixmap p;
    Cursor c;

    p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 10, 10, 1);
    c = XCreatePixmapCursor(dpy, p, None, &acolor, &bcolor, 0, 0);

    XFreeCursor(dpy, c);
    XDefineCursor(dpy, DefaultRootWindow(dpy), c); /*c is released!*/

    XFreePixmap(dpy, p);

    XCloseDisplay(dpy);

}
```

```
Resource 0x4a00002 of class 8 used, but is already released
   at 0x400EF18: XDefineCursor (xr_intercepts.c:402)
   by 0x80486E0: main (testDeInitializedCursor.c:15)
Release was at
   at 0x400EDE1: XFreeCursor (xr_intercepts.c:389)
   by 0x80486A2: main (testDeInitializedCursor.c:14)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Font: Use after release

```
#include <X11/Xlib.h>

int main(void)
{
    XFontStruct *fontinfo;
    Font f;
    Display *dpy = XOpenDisplay(NULL);

    f = XLoadFont(dpy, "fixed");
    XUnloadFont(dpy, f);
    fontinfo = XQueryFont(dpy, f); /*f is released!*/
    if (fontinfo)
    {
        /*XFreeFont automatically calls CloseFont on the fontid*/
        XFreeFont(dpy, fontinfo);
    }

    XCloseDisplay(dpy);
```

```
    return 0;
}
```

```
Resource 0x4c00001 of class 10 used, but is already released
   at 0x400F280: XQueryFont (xr_intercepts.c:469)
   by 0x80485EC: main (testDeInitializedFont.c:11)
Release was at
   at 0x400F13B: XUnloadFont (xr_intercepts.c:444)
   by 0x80485D8: main (testDeInitializedFont.c:10)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Pixmap: Use after release

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    XGCValues values;
    GC gc;

    Pixmap p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 200, 100, 1);

    values.foreground = WhitePixel (dpy, DefaultScreen (dpy));


    XFreePixmap(dpy, p);
    XFillRectangle(dpy, p, gc, 0, 0, 200, 200); /* p is released!*/

    XFreeGC(dpy, gc);

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 3 used, but is already released
   at 0x400E65C: XFillRectangle (xr_intercepts.c:282)
   by 0x8048710: main (testDeInitializedPixmap.c:15)
Release was at
   at 0x400EA07: XFreePixmap (xr_intercepts.c:333)
   by 0x80486CB: main (testDeInitializedPixmap.c:14)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Window: Use after release

```
#include <X11/Xlib.h>
```

```
int main(void)
{
    Display *dpy = XOpenDisplay(NULL);

    Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
        0, 0, 200, 100, 0,
        BlackPixel(dpy, DefaultScreen(dpy)),
        WhitePixel(dpy, DefaultScreen(dpy)));

    XSelectInput(dpy, w, ButtonPressMask | KeyPressMask);
    XMapWindow(dpy, w);
    XDestroyWindow(dpy, w);
    XSelectInput(dpy, w, ButtonPressMask | KeyPressMask); /*w is released!*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Resource 0x4c00001 of class 1 used, but is already released
   at 0x400E2FE: XSelectInput (xr_intercepts.c:236)
   by 0x80486D4: main (testDeInitializedWindow.c:15)
Release was at
   at 0x400E164: XDestroyWindow (xr_intercepts.c:209)
   by 0x80486B8: main (testDeInitializedWindow.c:14)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Colormap: Use before acquire

```
#include <X11/Xlib.h>

int main(void)
{
    XColor c; /*uninitialized*/
    Colormap map;
    Display *dpy = XOpenDisplay(NULL);

    XAllocColor(dpy, map, &c); /*c never initialized*/

    XFreeColormap(dpy, map);

    XCloseDisplay(dpy);

}
```

```
Uninitialised byte(s) found during client check request
   at 0x400F5F3: XAllocColor (xr_intercepts.c:521)
   by 0x8048568: main (testNeverInitializedColormap.c:9)
 Address 0xbed55214 is on thread 1's stack
 Uninitialised value was created by a stack allocation
   at 0x804853A: main (testNeverInitializedColormap.c:4)
 failed request:  BadColor (invalid Colormap parameter)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 22 from 2)
```

```
RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Cursor: Use before acquire

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    Cursor c; /*never initialized*/

    XDefineCursor(dpy, DefaultRootWindow(dpy), c); /*c never initialized*/

    XCloseDisplay(dpy);
    return 0;
}
```

```
Uninitialised byte(s) found during client check request
   at 0x400EF18: XDefineCursor (xr_intercepts.c:402)
   by 0x804855A: main (testNeverInitializedCursor.c:8)
 Address 0xbec60228 is on thread 1's stack
 Uninitialised value was created by a stack allocation
   at 0x804850A: main (testNeverInitializedCursor.c:6)
 failed request:  BadCursor (invalid Cursor parameter)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Font: Use before acquire

```
#include <X11/Xlib.h>

int main(void)
{
    XFontStruct *fontinfo;
    Font f; /*never initialized*/
    Display *dpy = XOpenDisplay(NULL);

    fontinfo = XQueryFont(dpy, f); /*f was never initialized*/
    if (fontinfo)
    {
        /*XFreeFont automatically calls CloseFont on the fontid*/
        XFreeFont(dpy, fontinfo);
    }

    XCloseDisplay(dpy);
    return 0;
}
```

```
Uninitialised byte(s) found during client check request
   at 0x400F280: XQueryFont (xr_intercepts.c:469)
```

```
 Address 0xbe805224 is on thread 1's stack
 Uninitialised value was created by a stack allocation
   at 0x804853A: main (testNeverInitializedFont.c:6)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Pixmap: Use before acquire

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    XGCValues values;
    GC gc;
    Pixmap px; /*never initialized*/

    Pixmap p = XCreatePixmap(dpy, DefaultRootWindow(dpy), 200, 100, 1);

    values.foreground = WhitePixel (dpy, DefaultScreen (dpy));
    gc = XCreateGC (dpy, p, GCForeground, &values);

    XFillRectangle(dpy, px, gc, 0, 0, 200, 200); /*px never initialized*/

    XFreeGC(dpy, gc);
    XFreePixmap(dpy, p);

    XCloseDisplay(dpy);
    return 0;
}
```

```
Uninitialised byte(s) found during client check request
   at 0x400E65C: XFillRectangle (xr_intercepts.c:282)
   by 0x80486F6: main (testNeverInitializedPixmap.c:15)
 Address 0xbea8f1b4 is on thread 1's stack
 Uninitialised value was created by a stack allocation
   at 0x80485DA: main (testNeverInitializedPixmap.c:8)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Window: Use before acquire

```
#include <X11/Xlib.h>

int main(void)
{
    Display *dpy = XOpenDisplay(NULL);
    Window w; /*never initialized*/

    XSelectInput(dpy, w, ButtonPressMask|KeyPressMask); /*w never initialized*/
```

```
    XCloseDisplay(dpy);

}
```

```
Uninitialised byte(s) found during client check request
   at 0x400E2FE: XSelectInput (xr_intercepts.c:236)
   by 0x8048538: main (testNeverInitializedWindow.c:8)
 Address 0xbef57224 is on thread 1's stack
 Uninitialised value was created by a stack allocation
   at 0x804850A: main (testNeverInitializedWindow.c:6)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 26 from 2)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Colormap: No Errors

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 21 from 1)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Cursor: No Errors

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 21 from 1)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Font: No Errors

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 21 from 1)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Pixmap: No Errors

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 21 from 1)

RESOURCE LEAK SUMMARY:
   definitely lost: 0 resources.
```

Window: No Errors

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 21 from 1)

RESOURCE LEAK SUMMARY:
```
definitely lost: 0 resources.

Appendix B: DSO Interposition

```
Pixmap (*real_XCreatePixmap)(Display *, Drawable , unsigned int ,
    unsigned int , unsigned int ) = NULL;

Pixmap XCreatePixmap(Display *display, Drawable d, unsigned int width,
    unsigned int height, unsigned int depth)
{
    Pixmap ret;
   if (!real_XCreatePixmap)
        *(void **) (&real_XCreatePixmap) = get_func("XCreatePixmap");
    ret = (*real_XCreatePixmap)(display, d, width, height, depth);
    restrack.acquired_resource(ret);
    return ret;
}

int (*real_XFreePixmap)(Display *, Pixmap ) = NULL;

int XFreePixmap(Display *display, Pixmap pixmap)
{
    int ret;
    if (!real_XFreePixmap)
        *(void **) (&real_XFreePixmap) = get_func("XFreePixmap");
    ret = (*real_XFreePixmap)(display, pixmap);
    restrack.released_resource(pixmap);
    return ret;
}

int XFillRectangle(Display *display, Drawable d, GC gc, int x, int y, unsigned
    int width, unsigned int height)
{
    if (!real_XFillRectangle)
        *(void **) (&real_XFillRectangle) = get_func("XFillRectangle");
    restrack.check_resource(ret);
    return (*real_XFillRectangle)(display, d, gc, x, y, width, height);
}

void restracker::acquired_resource(long nId)
{
    aAllocatedIds[nId] = backtrace...
    aActiveIds[nId] =  aAllocatedIds[nId]
}

void restracker::released_resource(long nId)
{
    aActiveIds.erase(nId);
    aReleasedIds[nId] = backtrace...
}

void restracker::check_resource(long nId)
{
    if (nId in aActiveIds)
        return; //No error
    else
    {
        fprintf(stderr, "invalid resource %d at", nId);
        show_location();
        if (nId in aReleasedIds)
        {
```

```
            fprintf(stderr, "Use of DeAllocated resource,"
                "resource was allocated at);
            aReleasedIds[nId].show_location();
            fprintf(stderr, "resource was originally allocated at);
            aAllocatedIds[nId].show_location();
        }
        else
            fprintf(stderr, "Use of Unintialized resource");
    }
}
```

Appendix C: Dedicated Valgrind Tool

```
Pixmap I_WRAP_SONAME_FNNAME_ZZ(libX11ZdsoZdZa,XCreatePixmap)(Display *display,
    Drawable d, unsigned int width, unsigned int height, unsigned int depth)
{
    unsigned int _qzz_res;
    Pixmap ret;
    OrigFn fn;
    VALGRIND_GET_ORIG_FN(fn);
    CALL_FN_W_5W(ret, fn, display, d, width, height, depth);
    restrack.acquired_resource(ret);
    return ret;
}

int I_WRAP_SONAME_FNNAME_ZZ(libX11ZdsoZdZa,XFreePixmap)Display *display,
    Pixmap pixmap)
{
    unsigned int _qzz_res;
    int ret;
    OrigFn fn;
    VALGRIND_GET_ORIG_FN(fn);
    CALL_FN_W_WW(ret, fn, display, pixmap);
    restrack.released_resource(pixmap);
    return ret;
}

int I_WRAP_SONAME_FNNAME_ZZ(libX11ZdsoZdZa,XFillRectangle)(Display *display,
    Drawable d, GC gc, int x, int y, unsigned int width, unsigned int height)
{
    unsigned int _qzz_res;
    int ret;
    OrigFn fn;
    VALGRIND_GET_ORIG_FN(fn);
    restrack.check_resource(ret);
    CALL_FN_W_7W(ret, fn, display, d, gc, x, y, width, height);
    return ret;
}

void restracker::acquired_resource(long nId)
{
    aAllocatedIds[nId] = backtrace...
    aActiveIds[nId] =  aAllocatedIds[nId]
}

void restracker::released_resource(long nId)
{
    aActiveIds.erase(nId);
    aReleasedIds[nId] = backtrace...
}

void restracker::check_resource(long nId)
{
    if (nId in aActiveIds)
        return; //No error
    else
    {
        fprintf(stderr, "invalid resource %d at", nId);
        show_location();
        if (nId in aReleasedIds)
```

```
        {
            fprintf(stderr, "Use of DeAllocated resource,"
                "resource was allocated at);
            aReleasedIds[nId].show_location();
            fprintf(stderr, "resource was originally allocated at);
            aAllocatedIds[nId].show_location();
        }
        else
        {
            //Use Valgrind Origin Checking to report undefined values
            VALGRIND_CHECK_VALUE_IS_DEFINED(nId);
        }
    }
}
```

Appendix D: DSO-side Of Hybrid Solution

```
Pixmap XCreatePixmap(Display *display, Drawable d, unsigned int width,
    unsigned int height, unsigned int depth)
{
    Pixmap ret;
    unsigned int _qzz_res;
    VALGRIND_DO_CLIENT_REQUEST(_qzz_res, 0, VG_USERREQ__USE_RESOURCE, &d,
        WINDOW | PIXMAP, 0, 0, 0); \
    ret = (*real_XCreatePixmap)(display, d, width, height, depth);
    VALGRIND_DO_CLIENT_REQUEST(_qzz_res, 0, VG_USERREQ__ACQUIRE_RESOURCE, &ret,
        PIXMAP, 0, 0, 0);
    return ret;
}

int XFreePixmap(Display *display, Pixmap pixmap)
{
    int result;
    unsigned int _qzz_res;
   VALGRIND_DO_CLIENT_REQUEST(_qzz_res, 0, VG_USERREQ__RELEASE_RESOURCE,
        &pixmap, PIXMAP, 0, 0, 0);
    result = (*real_XFreePixmap)(display, pixmap);
    return result;
}

int XFillRectangle(Display *display, Drawable d, GC gc, int x, int y,
    unsigned int width, unsigned int height)
{
    unsigned int _qzz_res;
    VALGRIND_DO_CLIENT_REQUEST(_qzz_res, 0, VG_USERREQ__USE_RESOURCE, &d,
        WINDOW | PIXMAP, 0, 0, 0);
    return (*real_XFillRectangle)(display, d, gc, x, y, width, height);
```

Appendix E: Using Origin Tracking

```
static void check_use_resource(ThreadId tid, UWord handle_addr, UWord idtype)
{
    /*Look up the handle_addr of resource type idtype in our tables of
      acquired and not released yet resource*/
    XR_Resource *resource = ...

    if (!resource) {
        /*This resource was never acquired, we need to use origin tracking
          to determine where this invalid value originated*/
        Addr bad_addr = handle_addr;
        UInt otag = 0;
        /*An example of priting the stacktrace of the current location*/
        VG_(get_and_pp_StackTrace) ( tid, 1000 );

        /*Extrac the origin tracking information for this invalid value*/
        if (MC_(clo_mc_level) == 3)
            otag = MC_(helperc_b_load1)( bad_addr );

        /*Use built-in reporting function to print stacktrace of where
          this value entered the program flow*/
        MC_(record_user_error) ( tid, bad_addr, /*isAddrErr*/False, otag );
    }
    else if (resource->release_context) {
        /* we recorded that this resource was released earlier*/
        VG_(get_and_pp_StackTrace) ( tid, 1000 );
        VG_(message)(Vg_UserMsg, "Release was at");
        VG_(pp_ExeContext) ( resource->release_context );
       VG_(maybe_record_error)( tid, 9000, 0, NULL, NULL );
    }
}
```

Appendix F: Annotated Bibliography

Bond, M.D., Nethercote, N., Kent, S.W., Guyer, S.Z., McKinley, K.S. (2007). Tracking bad apples: reporting the origin of null and undefined value errors. Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. ACM.

The authors present an approach to *origin tracking* of invalid values at run-time. An invalid value is typically seen in C/C++ where an uninitialized value has been referenced, or in Java where a value has been left accidentally null. A bug's effect is often far from its cause , and in 50% of cases of a crash due to such an invalid value, the method that introduced the invalid value is not in the immediate crash, the invalid value may have been propagated through the program by various assignments and other operations before being presented to the final method where the error becomes critical. Origin tracking enables automatically tracing through the flow of control to identify the initial location where the invalid value was introduced. The technique shown for C/C++ programs comes with the caveat that small values less that 32bits are not tracked, and that there is a small chance that incorrect origin information can be displayed, but that nonetheless "finds origins for 72% of the 32-bit undefined value errors". The ability to provide useful information about the *source* of a undefined value use is a key usability feature for such debuggers, and this paper provides a practical technique for achieving that goal.

Cantrill, B.M., Shapiro, M.W., Leventhal, H.L. (2004). Dynamic Instrumentation of Production Systems. Proceedings of the USENIX 2004 Technical Program. USENIX. Retrieved Feb 17, 2009 from http://www.usenix.org/event/usenix04/tech/general/full_papers/cantrill/cantrill_html/

This paper, by Sun Solaris kernel developers, describes the Solaris DTrace facility. DTrace (Dynamic Trace) is a kernel level extension whereby kernel level *providers* report as to what

instrumentation capabilities they could provide to the DTrace framework. A goal of DTrace is to have no run-time overhead while it is not enabled, and to be able to selectively enable reporting capabilities as requested to minimize the overhead of instrumentation on performance. DTrace providers include reports on entering and exiting function boundaries, lock and unlock primitives, system calls and time-based profiling callbacks. DTrace provides a D Language which users can use to specify arbitrary predicates and actions for each reported event with access to various parameters associated with the triggering event. DTrace shows an approach to selectively instrumenting a system to monitor particular events for the purposes of debugging system behaviour. The strength of DTrace is its ability to be scripted with D to combine together providers to form user-defined instrumentation directives and tracing infrastructure based on the built-in providers.

Curry, T.W. (1994). Profiling and Tracing Dynamic Library Usage via Interposition. Proceedings of the USENIX Summer 1994 Technical Conference. USENIX. Retrieved Feb 17, 2009 from http://www.usenix.org/publications/library/proceedings/bos94/full_papers/curry.ps

This technical paper describes the mechanics of *dynamic libraries* as found on UNIX and UNIX-like operating systems and how an additional dynamic library can be interposed between an application and the original destination library. A dynamic library or shared library is a collection of functions to which an application binds at run-time rather than at compile/link time. The linking occurs at execution-time and, as a side-effect of this, a replacement library can be interposed between that library and the application and replacement functions in the replacement library will be called instead of the original ones. The replacement functions may or may not forward the calls to the original library. The article describes various uses of the technique for logging parameter calls or recording call stacks for profiling purposes, e.g. an explicit example shown is that of recording X Windowing System library calls for profiling. The article provides a set of techniques for interposing instrumentation between an application and the libraries it uses at run-time without re-compilation of the application or libraries being instrumented.

Dumitran, D. (2007). Fixing File Descriptor Leaks. Master's thesis, Massachusetts Institute of Technology. Retrieved Feb 17, 2009 from http://dspace.mit.edu/handle/1721.1/41645

This thesis examines the problem of detecting and automatically fixing file descriptor leaks, to "design, implement, and test a mechanism of automatically closing leaked FDs, thus allowing applications which leak FDs to continue to operate normally". The problem of detecting file descriptor leaks is analogous to that of detecting server-side resource leaks, though different in terms that file descriptor leaks occur in the same address space as the application that causes them, but similar in that file descriptors tend to be a relative scarce resource, and a leak in one application will affect the availability of file descriptors for another application. The approach taken by the author is to use an interposed shared library to intercept C calls that utilize file descriptors to track the number of active file descriptors and to decide which file descriptor to force-able close when the offending application has exhausted the available descriptors. The thesis shows one mechanism for tracking resources allocated and deallocated through APIs by use of a interposed shared library, though the target is to provide a mechanism to avoid application failure on resource exhaustion rather than to provide a debugging tool to identify the location of the leaked resource.

Gettys, J., Scheifler, R.W. (2002). Xlib − C Language X Interface. X Consortium. Retrieved Feb 18, 2009 from http://ftp.xfree86.org/pub/XFree86/current/doc/PDF/xlib.pdf

This reference guide comprehensively documents the "low level C language interface to the X Window System protocol". It provides an overview the X Window System and detailed specifications of the API. It is the definite guide to the functions available in the libX11 library which acts as an intermediary between X Window applications and the layer which converts these calls into X Protocols messages which are passed to a remote X Server for eventual execution. Each argument available for each function is documented, and the possible errors that can be reported. Allocation routines for remote resources, e.g. XCreatePixmap, XCreateWindow are documented along with the routines which should be used to cause those remote resources to be released. The

handle types which refer to remote resources are documented as “integer resource IDs, which allows you to refer to objects stored on the X server. These can be of type Window, Font, Pixmap, Colormap, Cursor, and GContext”. This interface reference enables an analysis of the libX11 API to determine the complete set of methods called locally which affect the allocation and deallocation of remote resources in the X Window System.

Giraldeau, F., Dault, J.M, des Ligneris, B. (2006, September). MILLE-XTERM and LTSP. Linux Journal. Specialized Systems Consultants, Inc. Seattle, WA.

The authors describe MILLE-XTERM, a “scalable infrastructure for massive X-terminal deployment” based around the LTSP (Linux Terminal Server Project) offering where applications execute on a remote application server and display to a local X-terminal which provides only a X Server for display of graphics. The article provides an insight into the scalability of such a system, and provides a sample environment in which to appreciate the dangers of remote resource leaks as this environment is especially prone to X Server resource leaks and over-utilization, “For instance, several applications use the X-server memory as a cache memory. Although this is very efficient on a Linux workstation, it can cause an X-terminal crash when the memory used by the X server is bigger than the RAM of the terminal”.

Hastings, R., Joyce, B. (1992) Purify: Fast detection of memory leaks and access errors. Proceedings of the Winter USENIX Conference. USENIX. Retrieved Feb 17, 2009 from http://opera.cs.uiuc.edu/probe/reference/debug/dynamic/purify_92.pdf

This paper presents Purify, a commercial program that “developers and testers use to find memory leaks and access errors”. Purify is a DBA, dynamic binary analysis, tool that reports errors at run-time of the application being tested. Before execution the application is re-linked by purify in order to rewrite the binary to intercept attempts to read and write memory and tracks if an attempt to read/writer is on an invalid area of memory, or is a read to an uninitialized block. To track memory

leaks purify annotates every attempt to malloc memory with the address of the function that called malloc and then uses a variant of garbage collection to ascertain if a given block has been released. Purify was one of the pioneering debugging applications for successfully tracking memory resource leaks and errors, this paper provides a basis to compare future generations of tooling against.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J, Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. ACM.

This paper introduces the Pin system, a framework for building program analysis tools. Pin allows a tool writer "to analyse an application at the instruction level without detailed knowledge of the underlying instruction set". Pin is a run-time binary instrumentation system, the key strengths of Pin are that is is independent of the underlying instruction set from the perspective of the tool-writer, and it is relatively fast, i.e. "Valgrind slows the application down by 8.3 times, DynamoRIO by 5.1 times, and Pin by 2.5 times". Pin provides APIs to "observe all the architectural state of a process, such as the contents of registers, memory, and control flow". Although faster than Valgrind, Pin is targeted at effectively a lower level of granularity than Valgrind, focused on providing a fast framework on which tools can investigate the effect of relatively small amounts of code on cache performance and register usage. Writing a tool to deal with higher level call-level and value-tracking requirements is comparatively difficult. While the execution speed of Pin is superior to other offerings, the support for shadow variables to enable invalid value tracking is limited.

Maebe , J., Ronsse, M., De Bosschere , K. (2004). Precise detection of memory leaks. Second International Workshop on Dynamic Analysis. IEEE.

This paper addresses the problem of reporting the location where memory was lost in a memory leak. It is acknowledged that many tool exist which can report that memory was leaked,

and report where that memory was allocated, but that it is difficult to report where it was lost. This paper presents a technique that reports where the memory was allocated, where it was lost, and where it was last addressed. Leaks are categorized into two categories, those where the block is allocated and never freed, but a handle always exists to the block. And those where the handle to the block has been lost. The first termed a logical leak, and the second a physical one. The tooling described here tracks physical leaks only. By storing the location of each allocation and watching all memory to attempt to reference count references to those allocations (as opposed to the less resource intensive mark-and-sweep garbage collection mechanism typically employed by such tools) the approach described here enables reporting of where the handle to a memory resource was likely lost. The technique is heavyweight, but reporting the location of the loss of the final reference to a resource is an invaluable aid in debugging the problem.

Nethercote, N., Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. ACM.

This paper introduces the Valgrind framework for building debugging and program analysis tools. Valgrind is a dynamic binary analysis tool which functions at run-time, converting the binary code of the executable being examined into an intermediate representation, instrumenting that IR by tools that use the framework (e.g. the Memcheck tracks accesses to uninitialized values), and converting the IR back to executable format. One key relevant aspect is that “Valgrind supports function replacement, i.e. it allows a tool to replace any function in a program with an alternative function. A replacement function can also call the function it has replaced. This allows function wrapping, which is particularly useful for inspecting the arguments and return value of a function”. Which raises the possibility of reusing the Valgrind infrastructure to capture, inspect and interrogate the local calls to libraries that trigger remote services to allocate or use a remote resource handle similarly to how existing Valgrind tools handle kernel calls whose internal implementation is also

opaque to Valgrind, though for different technical reasons. Valgrind is presented as a framework which “makes tools relatively easy to write, allows them to be robust, provides powerful instrumentation capabilities, and allows reasonable performance”. This paper presents the capabilities of Valgrind and provides technical arguments for the choice of Valgrind as a framework on which to build new dynamic run-time debugging tools.

Nethercote, N., Seward, J. (2007). How to shadow every byte of memory used by a program. Proceedings of the 3rd international conference on Virtual execution environments. ACM.

The authors, designers and implementers of the Valgrind debugging framework, present a technique for for creating efficient dynamic analysis tools that shadow every byte of memory used by a program with another value that tracks certain information about that byte, e.g. how many times that byte has been accessed, or where it was initialized from. Shadow memory enables “tools that use it [to] detect critical errors such as bad memory accesses, data races, and uses of uninitialised or untrusted data”. The technical mechanism of implementation is shown, and performance compared against other similar implementations, to demonstrate the relative efficiency of the Valgrind approach. The capabilities of Valgrind to let a tool “remember something about the history of every memory location and/or value in memory” is a powerful aid to support tracking the origins of a value to determine e.g. if that value is the result of a procedure which caused a remote resource to be allocated or if the value presented to a deallocation procedure was already presented to such a procedure.

Nethercote, N., Walsh, R., Fitzhardinge, J. (2006). Building Workload Characterization Tools with Valgrind. IEEE International Symposium on Workload Characterization. IEEE.

This extensive tutorial on Valgrind introduces the Valgrind dynamic binary analysis and instrumentation framework. Among its features, this tutorial documents the abilities of Valgrind to replace arbitrary functions or wrap functions and crucially to track the value of any location in

memory or in a register, i.e. “Tools that shadow every register and/or memory location with a metavalue that says something about it”. The tutorial provides example of use of shadow values and provides the necessary documentation for implementing a new Valgrind tool which requires the ability both interpose between an application and shared library and to track a handle value through the life-time of an application to determine where it was originally initialized.

Rayside, D. Mendel, L. (2007). Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks . Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM.

This paper presents an approach to tracking a class of memory leaks. Memory leaks can be classified into two broad groupings, those where are no longer reachable (which can be detected in languages that support the principle by garbage collection and thus automatically released) and those which remain reachable (and so are not candidates for garbage collection) but are no longer required or used by the program thereafter, i.e. no longer “observably reachable” junk objects. The authors approximate detecting such junk objects by recording when an object has last been the target of a method call or has had its values read or written, an object which continues to exist but no longer affects the execution of the program becomes a stale object, and a junk object candidate. The technique used for Java object ownership profiling prompts consideration of applying the same technique to detect which scarce server resources in a client-server application which, while not leaked resources because the client retains a reference to them and releases eventually, could be released at a far earlier stage and returned to the pool of available server resources.

Scheifler, R.W., Gettys, J. (1986). The X Window system. Transactions on Graphics (TOG), 5(2). ACM.

This paper provides a comprehensive overview of the X Window system and documents the key design features. The X Window system is a client-server network transparent architecture, an

application running on one machine can display to another one, each physical display is managed by an X Server. As a consequence of the client-server nature and a desire for efficiency, certain basic resources are stored by the server and created and destroyed on request by the clients, “the basic resources provided by the server are windows, fonts, mouse cursors, and off-screen images. Clients request creation of a resource by supplying appropriate parameters; the server allocates the resource and returns a 29-bit unique identifier used to represent it”. It is acknowledged that clients are likely to forget to instruct the server to destroy a resource so “the maximum lifetime of a resource is always tied to the connection over which it was created. Thus, when a client terminates, all of the resources it created are destroyed automatically”. But clearly the design is one where application and the display resource manager are not within the same instruction space, or even necessarily on the same machine, and that un-released resources are retained for the life-time of the application. So a resource leaking long-lived application can exhaust the server of resources. This overview paper on the X Window system explains the architecture of the system and explains the life-cycle and location of the basic X Window resource types.

Srivastava, A., Eustace A. (2004). ATOM: a system for building customized program analysis tools. SIGPLAN Notices, 39(4). ACM.

ATOM, Analysis Tools with OM[timization System], is presented as a framework for building program analysis tools which provides the common instrumentation code required by such tools. Using ATOM information can be “directly passed from the application program to the analysis routines through simple procedure calls” and can be used for memory recording and profiling along with cache simulation, evaluating branch prediction and pipeline simulation. ATOM (like Pin and Valgrind) is intended to provide a framework that takes care of the details of binary instrumentation to allow a tool developer to focus on “what information is to be collected and how to process it”. ATOM however doesn't provide the higher level shadow memory that Valgrind provides, though it does provide detailed low-level mechanisms for accurate compiler and CPU-

designer profiling simulation measurements.