STatic (LLVM) Object Analysis Tool: Stoat

Mark McCurry
Georgia Tech
United States of America
mark.d.mccurry@gmail.com

Abstract

Stoat is a tool which identifies realtime safety hazards. The primary use is to analyze programs which need to perform hard realtime operations in a portion of a mixed codebase. Stoat traverses the callgraph of a program to identify which functions can be called from a root set of functions which are expected to be realtime. If any unsafe function which could block for an unacceptable amount of time is found in the set of functions called by a realtime function, then an error is emitted to indicate where the improper behavior can be found and what backtrace is responsible for its call.

Keywords

Realtime safety, static analysis, LLVM

1 Motivation

When using low latency audio tools an all too common problem encountered by users is audio dropout caused by an excessive run time of the audio generation or processing routine. This artifact is also commonly known as an xrun. Xruns can be generated when there's simply too much to calculate during the allocated time, but it can also be easily generated by any function which takes an unreasonable amount of real time to execute[1]. The latter category of functions typically include operations involving dynamic memory, inter-process communication, file IO, and threading locks.

For low latency audio to reliably work, a frame of audio and midi data must be processed within a short fixed time window. Audio callbacks are then known as functions bound by a real time constraint, or *realtime* for short. A large portion of code can have it's total execution time bounded when the size of data is known in advance. Some code however cannot be simply bounded. As a simple example, consider prompting the user for synthesis parameters and waiting for a response. The user could enter a response quickly or they could never provide a response. The class of functions which aren't bounded by the real time constraint, that a realtime program operates with, are known as *non-realtime*.

To avoid xruns, realtime programs should be composed of functions with reasonable realtime bounds, and thus non-realtime functions are unsafe for a reliable program. Typically realtime system programmers acknowledge the timing constraint and design systems with this limitation in mind. Simple tests may be used to identify the typical execution times as well as variance, though it's easy for bugs to creep in. In particular, the C or C++ open source projects in Linux audio frequently have architectural issues making realtime use unreliable[2].

Maintaining a large codebase in C or C++ can make it very difficult to both know what a given function can end up calling or when a particular function could be called. Typically problems start with a mixed realtime/non-realtime system, such as UI and DSP sections of code; the segregation within one codebase may not be at all clear in implementation. This is further complicated by the opaqueness of some C++ techniques, such as virtual overloading, operator overloading, multiple inheritance, and implicit conversions. Overall these complications make manual verification of large scale realtime programs difficult.

Stoat offers a solution to identifying realtime hazards through an easy-to-use static analysis approach. Static analysis makes it possible to identify when functions claimed to be realtime can call unsafe non-realtime functions even when complex C or C++ call graphs are involved. The approach offered by Stoat makes it easy to identify these programming errors which can be used to greatly improve the reliability of

[1] as opposed to cpu time

[2] see appendix A

low latency tools.

1.1 Prior Art

Stoat isn't the first tool to address the problem of identifying these realtime safety hazards. Several years prior to the creation of Stoat, Arnout Engelen created jack-interposer which is a runtime realtime safety checker [Engelen, 2012]. Jack-interposer works by causing a program to abort if within the JACK process callback any known unsafe non-realtime function is called. The functions which jack-interposer identifies as unsafe include, IO functions (`vprintf()`, and `vfprintf()`), polling functions (`select()`, `poll()`), interprocess communication (`wait()`), dynamic memory functions (`malloc()`, `realloc()`, `free()`), threading functions (`pthread_mutex_lock()`, `pthread_join()`), and `sleep()`.

As a runtime analysis tool jack-interposer requires the program to be executed to identify errors and each error is reported as it's encountered. Individual errors are presented by a message without a backtrace or by halting the program and allowing a developer to use a debugger. Jack-interposer has the same issue as other runtime tools compared to static analysis. Namely, exhaustive testing requires the user or testing script to run the program through all states which involve different logic. Doing so is a difficult, error prone, and tedious task. Additionally, jack-interposer was designed to only be used with JACK clients, while Stoat works with any program, JACK based or not.

Stoat is based of an earlier attempt at creating another more general static analysis tool. The predecessor project, Static Function Property Verifier, or SFPV, attempted to address more general problem of tracking described properties through a programs feasible call graph [McCurry, 2014]. SFPV used the Clang compiler's API to record precise source level information [Lattner, 2008]. Unfortunately the Clang API was subject to rapid breaking changes, slow to compile, and vastly underdocumented, so SFPV was rewritten to create Stoat. Stoat in comparison uses a limited subset of the LLVM API without interfacing directly with Clang.

2 Examples

Both runtime and static analysis tools, including jack-interposer and Stoat, attempt to address the same overall problem. Both aim at detecting when a function which can be executed in a realtime thread can call a function which may block for an unacceptable amount of time. In C, an example of this is is shown in listing 1. `root_fn()` can call `malloc()` through two intermediate functions, `unannotated_fn()` and `unsafe_fn()`. When Stoat is provided with an out-of-source annotation on `root_fn()` it can then use the call graph to deduce that an unsafe function can be called.

Listing 1: Example C Program

```
void root_fn(void) {
    unannotated_fn();
}
void unannotated_fn(void) {
    unsafe_fn();
}
void unsafe_fn(void) {
    malloc(10);
}
```

For a C program many call graphs are relatively simple and no complex type information is needed. C++ call graphs however make extensive use of operator overloading, templates, and class based inheritance. Listing 2 shows an example of the `root_fn()` calling a method which may or may not be safe based upon which implementation of `method()` is called. As the class hierarchy is available to Stoat, the root function can be conservatively marked as unsafe as `method()` would call `malloc()` if Obj was an instance of the Unsafe class. Depending upon the workload of a particular program, this data dependency might be satisfied very rarely, so a purely runtime based approach may not identify the error.

Listing 2: Example C++ Program

```
void root_fn(Obj *o) {
    o->method();
}
class Unsafe: public Obj {
    virtual void method(void) {
        malloc(10);
    }
};
```

3 Stoat Implementation

Stoat consists of several components. First, there is a compiler shim to dump LLVM based metadata though LLVM IR files. Second, there is a series of LLVM compiler passes to extract

inline annotations and call graph information. Last, there is a ruby frontend to perform deductions on the extracted call graph and to produce diagnostic messages and diagrams.

Stoat uses information present in LLVM bitcode to capture the program's call structure. Generating bitcode for individual files can be difficult to integrate with complex software projects' build systems. A similar issue was presented by Clang's official static analysis tools [Kremenek, 2008]. Their solution was to have a stand-in which replaces the normal C/C++ compiler[3]. Stoat offers two compiler proxy binaries, `stoat-compile` and `stoat-compile++`, which provide a way to simplify generating LLVM bitcode similar to Clang's scan-build toolchain. For an autotool based project analysing source code is as simple as running `CC=stoat-compile CXX=stoat-compile++ ./configure && make` and then running `stoat -r .`.

For each LLVM bitcode file Stoat runs four custom LLVM passes. These passes respectively identify: the function calls, or call graph within the program; the C++ virtual methods associated in each class; the C++ class hierarchy; and in-source realtime safety annotations.

First the call graph is constructed. Within the LLVM IR the Call and Invoke operations call another function and they contain metadata about what function is being called. For C functions this is relatively simple. Consider the IR associated with `void foo(){bar()}` in listing 3.

Listing 3: LLVM IR For C Call

```
define void @foo() #0 {
entry:
  call void @bar()
  ret void
}
```

For C++, extracting the call graph is somewhat more complex due to the introduction of virtual methods. Virtual methods are a structured version of function pointers calls and they can be identified by the two-step process to obtain the function pointer. First, a class instance is converted to the virtual function table, or vtable. Then, the method's ID is used to extract the method from the vtable and the resulting function pointer is called. The LLVM IR for a virtual call is shown in listing 5 and it corresponds to the source shown in listing 4.

Listing 4: C++ Call

```
void foo(void) {
    Baz *baz;
    baz->bar();
}
```

Listing 5: LLVM IR For C++ Call

```
define void @_Z3foov() #0 {
entry:
  %baz = alloca %class.Baz*, align 4
  %0 = load %class.Baz** %baz,
      align 4
  %1 = bitcast %class.Baz* %0 to
      void (%class.Baz*)***
  %vtable =
    load void (%class.Baz*)*** %1
  %vfn = getelementptr inbounds
      void (%class.Baz*)** %vtable,
      i64 0
  %2 = load void (%class.Baz*)** %vfn
  call void %2(%class.Baz* %0)
  ret void
}
```

Next the vtable calls need to be mapped back to real functions. Vtables are stored as a global symbols and can be identified by the "_ZTV" prefix used in normal C++ symbol mangling procedures. The class hierarchy can be reconstructed by identifying chained constructors from class to class.

With the information presented by the normal call graph and the C++ virtual methods an augmented call graph can be constructed. First, any vtable methods are assumed to call any method implementation of the base class or any child class. Then, suppression file entries are used to remove edges from the augmented call graph to avoid false errors typically seen in error handling.

The last LLVM pass looks for in-source safety annotations in the form of `__attribute__((annotate("realtime")))` and `__attribute__((annotate("non-realtime")))`. These annotations can be added to the end of a function declaration to add metadata to the function within the C or C++ source. The annotations are augmented with out-of-source annotations in the form of whitelist and blacklist files.

Once the augmented call graph is constructed and a subset of the functions in the program

[3] `http://clang-analyzer.llvm.org/scan-build.html`

are annotated, a series of deductions can be made. Any function which is called, but never implemented is assumed to be non-realtime if not specified otherwise. Any function which is unannotated and called by a realtime function is assumed to be realtime. Any function which is realtime or assumed realtime that calls a non-realtime function produces an error with an associated deduction chain.

The errors can be presented in either a textual or graphical form. The current format includes the function that calls the unsafe function along with the deduced path. An example of an error flagged by Stoat is dynamic memory use within jalv when it is in a debug mode.

```
Error #514:
serd_stack_new
##The Deduction Chain:
 - serd_writer_new : Deduced Realtime
 - sratom_to_turtle : Deduced Realtime
 - jack_process_cb : Realtime (Annotation)
##The Contradiction Reasons:
 - malloc : NonRealtime (Blacklist)
```

Alternatively, figure 1 shows a partial view of a graphical representation of call graph nodes involved in errors. When dealing with a legacy codebase the graphical representation tends to be preferable as it visually shows which routines contain the most errors, and which errors are the most common. Additionally, for C++ codebases who's error involve long template expansions the graphical representation shortens the displayed names to result in a still large, but more manageable view on the software's architecture.

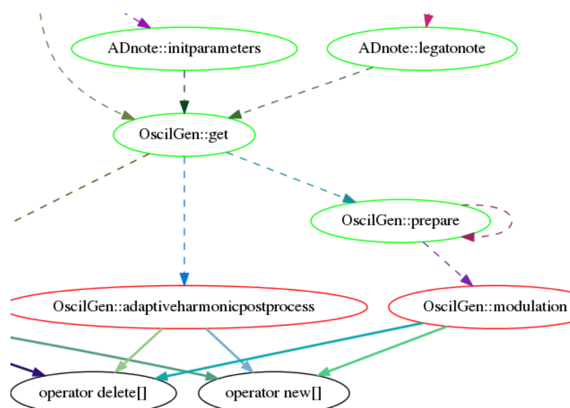

Figure 1: Partial output from Stoat applied to ZynAddSubFX 2.5.0[4]

[4] http://fundamental-code.com/2.5.0-realtime-issues.png

3.1 Limitations

Stoat offers a number of improvements over prior art, though Stoat does have its limitations. Namely, Stoat doesn't track data dependencies[5] on realtime safety. This task is one where runtime analysis tools, such as jack-interposer, can identify errors which Stoat isn't able to find or avoid false positives.

Two primary data dependent issues which produce misleading results include the use of unsafe function pointers and the use of unsafe error handling code. A short example of the former would be:

Listing 6: Function pointer call

```
void function(void (*fn)(void)) {
    fn()
}
```

If and only if `function()` is only passed realtime functions, then it is a realtime safe function, but the data passed into the function isn't analysed by Stoat, so function pointer calls are typically overlooked.

Debug and error handling code is a common source of false positives and the example error from jalv shows one such example. In listing 7, `function()` would be marked unsafe. The unsafe function should never be called in practical use and a runtime checker would not flag this case. A similar class of issues can occur if a function has different realtime safe behavior depending upon a flag passed to the function as may be the case with codebases which do not have separate functions for realtime and non-realtime tasks.

Listing 7: Example error handling

```
void function(void) {
    if(fatal_error)
        call_unsafe_function();
}
```

3.2 Discussion

Stoat and it's predecessor, SFPV, were originally created as a tool to assist with finding issues within the ZynAddSubFX synthesizer[6] and bringing it into compliance with realtime safety issues. While minor issues still exist, several users have reported improved reliability at lower latencies compared to earlier versions. Stoat has

[5] this includes any conditional code execution based upon constant or non-constant data

[6] http://zynaddsubfx.sf.net/

since been used as a verification tool in: librtosc[7], carla[8], ingen[9], and jalv[10]. Ideally it will be used on more projects within Linux Audio to identify realtime hazards in the future. The use of Stoat or jack-interposer would assist in correcting the poor user experience and possibly a negative reputation for stability that realtime hazards have created in a variety of realtime projects.

When Stoat doesn't understand regular structure within a program it is relatively easy to extend. ZynAddSubFX uses roughly 500 callbacks through librtosc. Stoat has already been extended to automatically annotate these callbacks As mentioned in the limitations, function pointers are difficult to reliably track with static analysis, librtosc callbacks however have per callback metadata which can be used to associate a statically known function pointer with information which can be used to identify which ones are expected to be executed in the realtime environment. This process was tested in ZynAddSubFX and used to resolve several bugs.

4 Conclusion

Stoat offers a new method to inspect existing software projects and direct attention towards code which may be responsible for realtime hazards. Addressing these realtime hazards can improve the experience within a variety of Linux audio applications and plugins. Through the use of automated tools such as Stoat realtime hazards can be identified and corrected quickly. Additionally, the static analysis approach of Stoat complements the prior art of runtime analysis that projects like jack-interposer provide. Stoat is available at `https://github.com/fundamental/stoat` under the GPLv3 license.

References

Arnout Engelen. 2012. Jack interposer. `https://github.com/raboof/jack_interposer`.

Ted Kremenek. 2008. Finding software bugs with the clang static analyzer. *Apple Inc.*

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar.

Chris Lattner. 2008. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2.

Mark McCurry. 2014. Static function property verification: sfpv. `https://github.com/fundamental/sfpv`.

A Brief Survey of Realtime Safety

In order to validate the claim that "projects in Linux audio frequently have architectural issues making realtime use unreliable", a survey was conducted on a sampling of Linux synthesizers. Each synthesizer as presented by `http://www.linuxsynths.com/` was given a brief manual code review (typically $<$ 15 minutes per project) looking for common realtime safety violations. If source code was not available or could not be located for a code review then the project was excluded. Projects marked with a '*' have had an in depth code review prior to the writing of this paper. The results shown in table 1 show that 18 of 40 projects (or 45%) have some easy to identify realtime safety issue.

Outside of LMMS and ZynAddSubFX the realtime hazards within each project has not received additional verification. Based upon experience working with projects not included in this list, additional realtime hazards are expected to be observed when tools like jack-interposer or Stoat are applied.

[7] `http://github.com/fundamental/rtosc`

[8] `http://kxstudio.linuxaudio.org/Applications:Carla`

[9] `https://drobilla.net/software/ingen`

[10] `http://drobilla.net/software/jalv`

Table 1: Linux Synthesizer Realtime Safety Observations

Software Name	Observed Status	Notes
6PM	likely unsafe	appears to launch threads within rt-thread
Add64	likely unsafe	blocking gui communication in rt-thread
Alsa Modular Synth	likely unsafe	unsafe data mutex
amSynth	likely unsafe	unsafe memory allocation in rt-thread
Borderlands	likely unsafe	unsafe locks/memory allocation in rt-thread
Bristol	likely safe	appears safe
Calf tools	likely safe	appears safe
Cellular Automaton Synth	likely safe	appears safe
Dexed	likely unsafe	unsafe memory allocations in rt-thread
DX-10	likely safe	appears safe
Helm	likely unsafe	memory allocation in rt-thread/addProcessor()
Hexter	likely safe	appears safe
JX-10	likely safe	appears safe
LB-302	likely unsafe	see LMMS
LMMS*	unsafe	unsafe locks in rt-thread, unsafe memory allocation in rt-thread, creation of threads in rt-thread, blocking communication to user interface in rt-thread, etc
Monstro	likely unsafe	see LMMS
Mr. Alias 2	likely safe	appears safe
Mx44	likely safe	appears safe
Nekobee	likely safe	appears safe
Newtonator	likely safe	appears safe
OBXD	likely safe	appears safe
Organic	likely unsafe	see LMMS
Oxe FM Synth	likely safe	appears safe
Peggy2000	likely safe	appears safe
Petri-Foo	likely safe	appears safe
Phasex	likely safe	appears safe
Samplev1	likely unsafe	possible memory allocation in rt-thread
SetBFree	likely safe	appears safe
Sineshaper	likely safe	appears safe
Sorcer	likely safe	appears safe
Synthv1	likely unsafe	possible memory allocation in rt-thread
Triceratops	likely safe	appears safe
Triple Oscillator	likely unsafe	see LMMS
Tunefish 4	likely safe	appears safe
Vex	likely safe	appears safe
Watsyn	likely unsafe	see LMMS
WhySynth	likely safe	appears safe
Wolpertinger	likely unsafe	unsafe memory allocation in setParameter()
Xsynth	likely unsafe	variety of mutexes used in the rt-thread
ZynAddSubFX*	unsafe	unsafe memory allocation in oscillator wavetable generation (the total number of realtime hazards was greatly decreased with the use of Stoat)