Reevers: Providing Non-Reentrance Immunity

Cristina Basescu
Automatic Control and Computers Faculty
University Poliltehnica of Bucharest
Email: cristina.basescu@cti.pub.ro

Razvan Deaconescu
Automatic Control and Computers Faculty
University Poliltehnica of Bucharest
Email: razvan.deaconescu@cs.pub.ro

***Abstract*—Non-reentrance immunity is a property by which a program, once afflicted by a failure due to a non-reentrant function, will attempt to provide a reentrant version of that routine. This is done so as to gain immunity to crashes that calling the non-reentrant routine would lead to. We propose Reevers, a program that detects whether an application written in C crashed while or after reentrantly calling a non-reentrant function. The detection process consists of a static and a dynamic phase attempting to improve accuracy, while keeping the result false negative free. We further propose some techniques for achieving immunization, making Reevers a complete tool.**

I. Introduction

A function is reentrant if, while it is being executed, it can be reinvoked by itself or by any other routine, without producing data corruption. By contrast there is the definition of a non-reentrant routine as one that is not recursive and cannot be shared by more than one task unless mutual exclusion is ensured either by using a semaphore or by disabling interrupts during critical sections of code.

In single-threaded programs, there is only one control flow, so the only possibility a routine could be re-entered is when it is recursively called. The programmer is usually aware of this situation, so this is an unlikely bug cause. However, the situation is different concerning concurrent programs, as it is more difficult to take into account all possible thread interleavings. In this case, the application would not provide consistent results and also may crash. Another aspect that could cause problems is signal generation and handling.

The problem that arises is that non-reentrant routines appear frequently in programs. A typical example is a code that was ported either from a platform that protected the process from this kind of bug or from a single-threaded environment to a multi-threaded one. Also it might be the case of using third party libraries which contain non-reentrant routines.

Moreover, the kind of bugs that non-reentrant routines develop is difficult to detect, since they may not appear until the code is executed by a large number of threads on a multi-threaded machine. However, not all non-reentrant routines generate bugs, some of them might always be executed in a safe manner. Others could simply produce erroneous output without crashing, thus it is beyond the purpose of this paper to detect all such routines.

In this paper we introduce the notion of non-reentrance immunity, which enables failure detection caused by non-reentrance and provides a base for non-reentrance immunization. In the rest of the paper present related work (II), provide an overview of Reevers, a mixed static and dynamic analysis tool for non-reentrance detection (III), describe the analysis phases (IV), evaluate it (V) and finally future work and conclusions are discussed in (VI).

II. Related Work

Taking into account the fact that we are heading to a parallel world, important research efforts are underway in the field of concurrent programming, especially regarding race conditions and atomicity. The main reasons for employing parallelization include achieving low latency and high throughput and scalability [4]. However, we believe reentrancy is at least as important as the issues mentions before, as it not only ensured thread safety, but also provides secure signal handling and recursion.

The paper [5] presents a taxonomy of concurrent bug patterns, which they use to create timing heuristics so as to increase the probability for concurrent bugs to occur. Their tool, ConTest, helps debugging concurrent programs by combining a replay algorithm with different thread interleavings, altered by the heuristics mentioned above. Although their solution is useful in finding bugs before they manifest, it doesn't offer alternatives for protection against them.

Atomicity is a stronger property than race conditions when it comes to ensuring correctness of concurrent programs [6]. The authors developed an instrumented semantics that checks whether the current execution and also some other similar executions are reducible to an equivalent serial execution i.e. they search for evidence of atomicity violations. Although their dynamic approach also works without programmer annotations, in order to reduce false alarms they eventually need that. Implementation and testing focus on Java programs.

The problems that arise from signal handling are pointed out in [7], where the authors propose a purely static technique to detect them. Also [8] suggests the problem of lacking well defined protocols for device drivers should be addressed by using a formalism made of state machines. We find this approach prohibitive, as programmers usually do not want to add anything else to their code.

III. Reevers Overview

In this section, we define the problem and present the goals (II.A), describe the high level architecture to accomplish them and shortly review the components (II.B) and illustrate how it works on an example (II.C).

A. Problem Definition and Goals

Programs augmented with non-reentrance immunization develop antibodies to prevent them failing i.e. crashing, due to previously encountered non-reentrance bugs. This paper addresses the first phase in providing this kind of immunity, more specifically non-reentrance detection. Such a technique should satisfy some major requirements. First, it should have a low false positives rate, i.e. not mistakenly detect a crashing cause as being a non-reentrance issue. Registering many false positives would cause high unneeded immunization overhead, which would further lead to unnecessary slowdown. Second, no false negatives are accepted in the detection phase i.e. it should detect all non-reentrant routines that lead to crashes; this way, the program's resistance to non-reentrance is surely augmented in time.

The main objective of Reevers is to provide coarse grained i.e. function level immunization against non-reentrancy, which by definition also provides thread safety. This should ensure correct program behavior offering a solution fast, but for performance enhancement further modifications by hand may be needed e.g finer grained immunization depending on the failure's cause. However, as immunization supposes first registering a crash, approaching safety-critical systems is not one of Reevers' goals.

B. System Architecture

In order to provide non-reentrance which lead to crashes detection for general-purpose applications, we employ both static and runtime analysis techniques. Not only does this hybrid detection technique guarantee a low false positive rate, but also ensures no false negatives.

Figure 1 illustrates the high-level architecture of Reevers. The system has two main entities: a component which instruments the target application, represented by the red block in the figure, and some gray blocks, which do not interact with the target application while it is running. Furthermore, the latter can be classified into blocks that are executed before or after the target application runs; we call the first class *before analysis* and the second one *after analysis*.

As shown in the figure, the Aspect Oriented Programming (*AOP*) block instruments the target application while running and saves in a file the call stacks of the threads that executed the application, namely *Threads Call Stacks* file. However, a reason for not using a core dump at this step is that we are interested in all the functions the threads called, not only in the ones that were still executing when the application crashed. Also, a second reason is that our realm of interest extends upon the order of function calls and their interleaving in different threads. Switching to *Static Analysis* block, its role is to determine which routines from the application are non-reentrant and might lead to crashes, provided that it has as input a set of already known non-reentrant library functions, e.g. *printf* from *stdio.h*, *gethostbyname* from *netdb.h* or *strcpy* from *string.h*. The input and the output are stored in the files *Potentially Non-Reentrant Functions* and *Library Non-Reentrant Functions* respectively. Making the initial assumption that the application does not crash, *Static Analyzer* will first run after such a crash was encountered. However, as this component needs to be executed just once, it will be further available to all subsequent crashes, as a result we can consider it a *before analysis* component. Finally, if the application finishes abnormally e.g. crashes, we apply at least one *post analysis* task. Having the *Threads Call Stacks* and the *Potentially Non-Reentrant Functions* files available, we first apply *Analyze While-Call Crash*, which determines whether the function that was executing when the crash happened was actually non-reentrant, but used as reentrant by the application at runtime. In addition, if this step does not provide cogent results, we employ a second *post analysis* component, *Analyze Post-call Crash*. This is usually necessary, because the common case for a non-reentrant function is to cause problems after its call ended, e.g. due to data corruption. More specifically, this step attempts to find a non-reentrant function reentrantly called sometime in the application's runtime history that could have caused the function where the application crashed to have this behavior.

C. Example

We present here a sample program which crashes due to non-reentrancy issues, that we will use as an example to illustrate the interaction between components in Figure 1.

```
int *v[DIM];
int length = -1;

void baz() {
    int i;
    for(i = 0 ; i < DIM ; i++)
line1: printf("line %d ", *v[i]);
}

void bar(void *matrix) {
    /* make a new location in v point to a line
       of the liniarized matrix */
    length++;
line 2: sleep(1);
    v[length] = (int*)(matrix + length * DIM);
    // when v is filled, call baz
    if(length == DIM-1) baz();
    else bar(matrix);
}

void *foo(void *matrix) {
    sleep(1);
    bar(matrix);
    pthread_exit(NULL);
}

int main() {
    int matrix[DIM*DIM];
    // all locations in v point to NULL
    // initialize matrix
    // create threads
    for (i = 0 ; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, foo,
      (void *)matrix);
    for (i = 0 ; i < NUM_THREADS ; i++)
        pthread_join(tid[i], NULL);
}
```

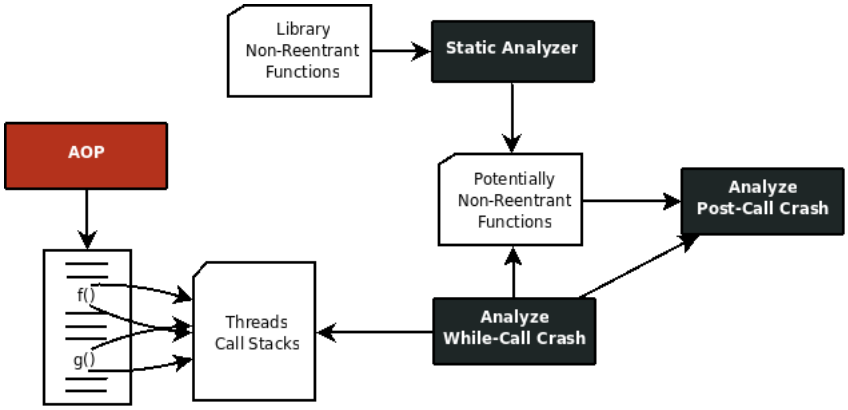

Fig. 1. High-level architecture of Reevers

The code fragment above shows a sample C code including a non-reentrancy bug, where two threads cooperate to make a vector's locations point to the lines of a matrix. *Static Analyzer* determines that function *bar()* is non-reentrant, as it uses global variables *v* and *length*, and also function *baz()* is non-reentrant, using global variable *v* and non-reentrant library function *printf*. At runtime, threads t_1 and t_2 simultaneously enter functions *foo()* and *bar()*, due to the *sleep()* call in both of the functions. Moreover, *bar()* is possibly reentered by the same thread, as both threads cooperate and recursively call *bar()* to fill vector *v*. However, due to the *sleep()* call from $line_2$, some locations from *v* would not be initialized and would still be pointing to *NULL*, therefore causing a SIGSEGV at $line_1$ in *baz()*. Besides this pieces of information, *Threads Call Stacks* also states that at runtime *baz()* will sometimes be reentrantly entered, but not always; all these are saved in the file mentioned above as $\langle threadId\ enter/exit\ functionName\ timestamp \rangle$. In the former case, *Analyze While-Call Crash* will detect *baz()* as responsible for the crash, as it is non-reenrant, reentrantly called and receives SIGSEGV. However, in the latter case, further analysis is needed from *Analyze Post-Call Crash*, which would see that *bar()* was previously reentered without being reentrant and, more important, it uses the same global variable as *baz()*, namely *v*, so it will state *bar()* as being the crash cause.

IV. Design

This section describes in depth the detection technique used by Reevers. First, we provide an overview on the tools used to cope with the challenges (III.A) and then we present the algorithms employed by each component (III.B). We also propose an immunization technique (III.C) for avoiding non-reentrant functions that lead to crashes, leaving the implementation for the latter as future work.

A. Tools

In order to employ static analysis in our tool, we use Clang[1], a C language family frontend for LLVM. LLVM[2] is a compiler infrastructure, containing a compilation strategy, a virtual instruction set, a compiler infrastructure, suitable for language-independent analyses and optimization and extensive interprocedural analysis, amongst others. LLVM's static analysis access is done at a low level e.g. load and store instructions, therefore we used Clang, which creates a new C, C++, Objective C and Objective C++ front-end for the LLVM compiler, at a higher level. Of interest for us was the support for static analysis clients, providing access to instructions and all their successors and predecessors, as well as finer grain access to the components of an instruction and their typer e.g. the variable declared and its storage type. These are compiled in so called clang plugins, basically shared objects, which are applied on the target application using *clang-cc* compiler.

Moving to the runtime analysis, we chose *Aspect Oriented Programming* so as to populate *Threads Call Stacks* file, more precisely the ACC[3] implementation. The next few lines provide a short introduction in AOP. The key concepts of AOP are aspects and advices. AOP uses load-time weaving in order to add functionality, which is represented by an advice. An aspect refers to certain places in the code that are interesting for the problem. AOP needs to know where exactly to apply the advices and this can be achieved by identifying arbitrary events in the runtime system. There are several reasons for which I think AOP is the best choice. First, it is a lightweight solution for solving the problem in a non-intrusive manner and second, aspects can be deployed and undeployed during runtime, which is useful for adding functionality on-the-fly. ACC requires access to the target application's source files, but we consider this not being a problem, as Reevers' intended usage is to help developers cope with non-reentrance problems before shipping applications to clients.

B. Algorithms

A reentrant routine either uses local variables or protects its data when global variables are used. There are some conditions a reentrant function should meet [9], [10], [11], [12]:

- It doesn't hold static data over successive calls
- It does not return a pointer to static data; all data is provided by the caller of the function
- It uses local data or ensures protection of global data by making a local copy of it

- It must not call any non-reentrant functions

Static Analyzer analyzes each statement from each function, searching for breaches of at least one of the rules mentioned above and stores them, as we already mentioned, in *Potentially Non-Reentrant Functions* file, one function name on each line. Regarding the last rule, it is applied only to those functions that call a function whose name is in *Library Non-Reentrant Functions* file, which also stores one function name on a line. To clarify this, we take as example a function $f_1()$, which calls non-reentrant function $f_2()$. If there are no other reasons for $f_1()$ to be non-reentrant other than calling $f_2()$, then a crash might appear only because $f_2()$ is reentrantly entered, so it is enough to store $f_2()$ as non-reentrant. However, if $f_2()$ is a library function, this action makes sense because for the future immunization step, described in the next subsection, it is easier to modify $f_1()$, whose source files ACC has access to.

Moving on to *AOP* code, the aspects consist in identifying the *before/after execution* of each function events, with the advices of storing into *Threads Stack Traces* file the id of the thread performing the operation, a value meaning whether it enters or exits the function, the name of the functions and a timestamp. In addition, another aspect/advice pair is needed in order to register a signal handler for SIGSEGV before executing *main()*. Its role is to write in an auxiliary file the id of the thread that received the segmentation fault signal, so as to be able to distinguish which of the last functions from each stack trace actually received SIGSEGV.

After the program finishes execution, if the exit code indicates that an error had occurred, *Analyze While-Call Crash* is the first *post-analysis* component that tries to detect whether the failure was caused by a non-reentrancy issue. First, it sorts the entries from *Threads Call Stacks* file by timestamp value, keeping them in memory as a map having as key a thread id and as value a list of $\langle enter/exit\ functionName\ timestamp \rangle$ pairs, and then applies an algorithm similar to finding nested closed intervals. For instance, if thread t_1 enters *f()* at $time_{11}$ and exits it at $time_{12}$, with $time_{12} > time_{11}$ and thread t_2 enters *f()* at $time_{21}$ and exits it at $time_{22}$, with $time_{22} > time_{21}$ and the intervals [$time_{11}$,$time_{12}$] and [$time_{21}$,$time_{22}$] intersect, then *f()* is reentrantly called. Notice that in the case of a recursive call, $t_1 = t_2$. Moreover, the names of these reentrantly called functions will be stored in an auxiliary file, so that they are also available for *Analyze Post-Call Crash*, if needed. Next, *Analyze While-Call Crash* will determine the function that was interrupted by the segmentation fault signal, looking at the last called function by the thread whose tid is stored in the auxiliary tid file mentioned in the previous paragraph. Finally, if this function is reentrantly called, but is also found in *Potentially Non-Reentrant Functions*, the detection phase ends here. Otherwise, *Analyze Post-Call Crash* will detect which of the reentrantly called functions from the auxiliary file, that also are non-reentrant i.e. stored in *Potentially Non-Reentrant Functions*, use the same global variables as the function that received SIGSEGV. The reason for doing so is that those global variables may be left in an inconsistent state by the non-reentrant function, which further causes the receival of segmentation fault. Consequently, these functions are issued by the detection phase.

C. Immunization Technique

We have thought of an approach to generate and apply a reentrant form for the function detected by the previous step. In order to do this, the detection phase should put in a table all the non-reentrancy reasons for the functions it outputs. In addition, a table with already known reentrant forms should be available, which usually contain a "_r" added to the function name in the nonreentrant form [12]. Examples are most of the string functions in C, like *strtok* and *strtok_r* from *string.h*.

This phase may imply multiple attempts in order to reach the reentrant form, if this is possible. Every attempt means also holding some rollback information, such as the old non-reentrant function, so as if several attempts are necessary, rollback can be performed so each attempt can start form the original state of the program.

A first choice that has to be made is whether to modify the function's interface or not. Although modifying the interface so as the caller provides all the data the function works on should be good in most cases, this also means a lot of changes in the source code and backward incompatibility. Consequently, we decided not to modify the interface when another solution exists. If the interface is changed, we should keep a copy of the old signature of the function, the reason for this being explained in the subsubsection *Avoiding Non-Reentrant Routines*.

In the first attempt, we generate the reentrant form by using some heuristics:

- If the bug's cause is the fact that the function returns a pointer to static data, a solution is to return dynamically allocated data instead, which holds the same information as the static one. It is the caller's task to free the memory.
- If the function uses static variables to keep data over successive calls, the interface needs to be changed. The caller should use a local variable that holds the same data as the static variable and pass it to the function.
- All global variables that the function uses should be declared volatile.
- If the bug is caused by a function that relies on singleton resources, the access to those resources should be serialized. A way to do that is by using mutexes. However, this approach is more likely a thread-safety one and does not guarantee the result.
- Sometimes signals can cause non-reentrant behavior of the functions, so the patch is that the caller saves the current set of signals and mask the signal set with the unwanted signals before it makes the call. After the call returns, the caller resets the signal set.

The second attempt, which is performed when the first attempt also leads to bugs or it was not possible to be applied (e.g. the source code wasn't available), is a more time consuming one. The solution is that a thread rebuilds the state of the program before the first entrance in the non-reentrant

function, based on a richer in information *Threads Call Stacks* file (which should also provide line numbers) and then calls the function to see what its effect should be. In oder to build a reentrant function that has exactly the same effect, we suggest using the functional programming paradigm, as it uses only referentially transparent functions, which are also reentrant, and then convert the code into our language, C. The advantage of this approach is that it gives information on the cause of the bug. However, the tradeoff is that the patch cannot be automatically generated.

1) Avoiding Non-Reentrant Routines: We find Aspect Oriented Programming to be the best choice here as well, having as aspects functions' name and as advices the reentrant form, kept in a table. If the solution consists of more pairs of aspect/advice, they should only be applied together.

An important thing is that the generated patch applies only in the place that generated the bug i.e. the line numbers issued in *Threads Call Stacks* file. As a matter of fact, it would be inefficient to search all the calls of that function in order to generate aspects for them. Moreover, sometimes this is not necessary, as there may be some calls that are executed in a guaranteed single-threaded manner. An interesting aspect is that even though we have patched a function and it does not fail in a given amount of time, rollback information still needs to be kept, as bugs may still appear later.

V. Evaluation

We present in this section several non-reentrancy patterns detected by the *Static Analyzer*. Below there is a code excerpt having some non-reentrancy issues, which become obvious when imagining a thread enters function *f()* when another thread is already executing it and has reached different points inside it.

```
static int a = 2;

int* f(int* param) {
1: static int z = 7;
2: z += *param;
3: int *w = &z;
4: a += *w;

5: int y = *param;
6: if(y==7)
7:     printf("y points to %d\n", *y);

8: w = param;
9: *w = 2;

10: if(a == 2)
11:     return &z;

12: return NULL;
}
```

- At lines 1, 2 and 11 static variable z is incremented with the value of *param* and then its address is returned. There are several problems here. First, *param* is possibly a reference parameter and we cannot be sure what value it will point to; as a matter of fact, at lines 8 and 9 we can see *param* changing its initial value to 2 and if some other thread does this while the current thread already entered *f()* but didn't reach line 2, z will have another value than the one expected. Second, returning z's address means several calls to *f()* will store their results at the same location, overwriting anything was there before.
- At lines 3 and 4, the value of the global variable a is changed to z's value, which, as explained earlier, is uncertain, so at line 10 the comparison will almost certainly not do what it should do. Moreover, a's value when entering *f()* depends on how many times the function was entered before.
- At lines 5 and 6 we use *param*'s value to make a comparison, but again *param*'s value may not be the one intended, as explained earlier i.e. due to it's change at lines 8 and 9.
- At line 7 the non-reentrant function *printf* is used, making *f()* also non-reentrant.

All these problems lead to receiving SIGSEGV when any of the variables implied would, due to an arbitrary thread interleaving, point to NULL. For instance, if the caller of *f()* stores z in a variable *result1*, returning at line 11, and from another thread another caller does the same with *result2*, but the function return at line 12 NULL, then using the value pointed by *result1* would receive segmentation fault. This situation is detected by *Analyze Post-Call Crash*.

VI. Future Work and Conclusions

In this section, we first introduce the future work, with some remarks on signal handling extending the ideas presented in III.C, and then conclude.

The special problem that signals pose is that the handler can be called asynchronously and this could happen, for example, when the process was executing a non-reentrant function or processing global data. Some heuristics may prove useful here e.g. making global variables accessed by a handler volatile. However, it is rather probable that human intervention will be needed in order to modify the handler, so the best effort here, besides applying the heuristic described in section III.C, is to detect the handler that caused the bug. This can be done by inspecting the *Threads Call Stacks* file, which should also keep the executed signal handlers.

In *Analyze Post-Call Crash* it would be interesting to detect the connection between the functions using either the current restriction, to use the same global values, or using variables that alias. However, we shall see whether the overhead brought by performing an alias analysis is worth. Another interesting idea is to make a difference between *using* a variable and *modifying* it. For example, if a function just uses the value of a global variable, but does not modify it, then this is not a reason for classifying it as non-reentrant. The problem here is that these *modify chains* might have a significant computational overhead, especially when pointers are used.

We have presented Reevers, a non-reentrance immunity tool, that targets deadlock bugs in general-purpose applications. First, it statically detects possible non-reentrant functions. Then, Reevers uses runtime information to detect if a crash

is due to non-reentrancy issues. The proposed detection technique is fast and fairly accurate, but it is not throughly tested as it seems to be a challenge to find applications that fit Reevers' target. However, enhancing Reevers with immunization capabilities will make it an attractive tool and broaden the spectrum of application Reevers can be applied on, for instance for replacing library non-reentrant function with their already known library reentrant versions.

REFERENCES

[1] (2009) clang: a C language family frontend for LLVM [Online]. Available: http://clang.llvm.org/

[2] (2009) LLVM: Low Level Virtual Machine [Online]. Available: http://llvm.org/

[3] (2008) ACC: The AspeCt-oriented C compiler [Online]. Available: http://research.msrg.utoronto.ca/ACC

[4] D, Dig; J. Marrero; and M. D. Ernst, *How do programs become more concurrent? A story of program transformations*, Report from Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2008

[5] E. Farchi; Y. Nir; and S. Ur, *Concurrent bug patterns and how to test them*, In IPDPS 03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, page 286.2. IEEE Computer Society, 2003

[6] C. Flanagan and S. N. Freun, *Atomizer: A dynamic atomicity checker for multithreaded programs*, In Annual Symposium on Principles of Programming Languages: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Venice, Italy , Pages: 256 267, 2004

[7] J. R. Diamant; W. Satterfield; and K. C. Wong, *US Patent 5822589 - Method for locating errors in a computer program*, 1998

[8] L. Ryzhyk; T. Bourke; and I. Kuz, *Reliable device drivers require well-defined protocols*, Proceedings of the 3rd workshop on on Hot Topics in System Dependability, p.3-es, Edinburgh, UK, 2007

[9] (2008) IBM Systems Information Center [Online]. Available: http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic= /com.ibm.aix.genprogc/doc/genprog/writing_reentrant_thread_safe_code.htm

[10] (2008) IBM developerWorks [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-reent.html

[11] (2008) The GNU C Library [Online]. Available: http://www.delorie.com/gnu/docs/glibc/libc_493.html

[12] (2008) SUN Developer Network [Online]. Available: http://developers.sun.com/solaris/articles/multithreaded.html