A Flexible Generator Architecture for Improving Software Dependability

Christof FETZER, Zhen XIAO
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
{christof, xiao}@research.att.com

Abstract

Improving the dependability of computer systems is increasingly important as more and more of our lives depend on the availability of such systems. Wrapping dynamic link libraries is an effective approach for improving the reliability and security of computer software without source code access. In this paper we describe a flexible framework to generate a rich set of software wrappers for shared libraries. We describe the architecture of the wrapper generator, the problems of how to generate wrappers efficiently, and our solutions to these problems. Based on a set of properties declared for a function, the generator can create a variety of wrappers to suit the diverse requirements of application programs. Performance measurements indicate that the overhead of the generated wrappers is small.

Keywords: *wrappers, wrapper generator, software fault-tolerance, middleware, reliability, security.*

1. Introduction

With the growing importance of computer applications in our society, the reliability of computer programs has received a lot of attention. Previously replication mechanisms targeted towards increasing the reliability of distributed services have been intensively investigated in the literature. For example, in the state machine approach [11] a service is replicated among a group of servers that process each request in lock step. The service remains available even if some servers in the group crash. This approach works well if the failures of different servers are *independent*. For example, power failures are independent if different servers have an independent power supply/backup. In addition, certain hardware failures can be made independent by running a service on different types of machines.

On the other hand, software failures are usually not independent and cannot be handled through replication [7]: if there is a bug in the software, the bug is likely to be replicated to all servers in the group. In recent years the complexity of software programs has increased dramatically. Therefore, an effective approach to handle software failures is critical to improving the reliability of distributed service.

One way to enhance the correctness of computer software is through formal methods. In this approach, the properties of a computer program are described by a set of specifications. An example of a specification for a distributed service can be that all messages received must be delivered in FIFO order. The correctness of an implementation is checked against its specification through formal analysis. This can give strong guarantees regarding the behaviors of the program in various execution contexts.

However, formal methods cannot address all problems related to software robustness. The specifications of a program usually abstract away many implementation details that need to be considered in practice. In fact, such a program often needs to be written in some special, *safe* language amenable to formal analysis and manipulation. Unfortunately, the majority of software (including most operating systems) today are written in unsafe languages like C or C++ that cannot be verified using formal methods. Moreover, the source code of commercial software is usually not available.

In this paper, we describe a novel generator approach to increasing software reliability through fault containment wrappers. The wrappers are generated automatically by a wrapper generator for dynamic link libraries in C . The generated wrappers can be used to detect various software failures without access to the source code. They can be used to collect statistics regarding failure types and causes in popular applications. They can also be used to provide appropriate debugging and failure diagnosis support. The wrapper generator is developed as part of the HEALERS project which is targeted towards increasing the robustness of applications.

The rest of the paper is organized as follows. Section 2 introduces the architecture of the wrapper generator. Section 3 describes a retry wrapper that can retry failed function calls due to transient problems in the system. In Section 4 we discuss implementation issues one has to address when creating a wrapper. Section 5 provides careful measurements of the overhead of the generated wrappers. Related work is described in Section 6. Section 7 concludes this paper.

```
Wrapper:
  void exit(int status) {
    abort() ;
  }

Start wrapper
 > setenv LD_PRELOAD ‘pwd‘/wrapper.so

Test it:
 > date
 Wed Oct 18 13:06:41 EDT 2000
 Abort
```

Figure 1. An example wrapper that overwrites `exit` by `abort`.

2. Architecture

We have designed and implemented a system called HEALERS (HEALers Enhanced Robustness and Security). The goal of our system is to increase the robustness of applications even if the source code is not available. This is achieved through a set of dynamically loadable **C** library wrappers that perform careful error checking for every function call to the **C** library. On most UNIX systems a user interested in using our wrapper can preload it by defining the `LD_PRELOAD` environment variable. When a program executes a function, it invokes the version of the function in our wrapper which can then perform error checking, usage metering, or various other tasks.

A very simple wrapper is illustrated in Figure 1 for the `exit` function. In **C** the `exit` function causes a program to terminate normally. However, sometimes we find that a program terminates unexpectedly and would desire some debugging information regarding the execution context when the program terminates. Therefore, we wrapped the `exit` function so that it invokes the `abort` function instead. This generates a *core dump* file when the program terminates which contains useful debugging information such as the stack trace. A nice feature of this approach is that it requires no source code modification or recompilation.

2.1. Wrapper Generation Steps

The generation of wrappers is performed in two steps (see Figure 2): the first step is performed by the *fault-injection generator* and the second step by the *wrapper generator*. The fault-injection generator derives the prototypes of the functions in a library by parsing header files and manual pages in the library. To generate wrappers, the wrapper generator might need to know additional properties of the wrapped functions. For example, a robustness wrapper increases the robustness of libraries by checking that the arguments of a function call are “robust”, i.e., do not result in a crash of the function. To derive some of the needed properties, our system performs automated fault-injection experiments. For example, we determine robust argument types for a function using fault-injection experiments. We call the prototypes of the functions in a library combined with other properties like the robust argument types *function declarations*. More details on the first step can be found in a companion paper [4]. The function declarations are then used by the wrapper generator to generate a variety of wrappers (e.g., robustness or security wrappers) in the second step. In the remainder of this section, we describe the function declarations and the architecture of the wrapper generator in more details.

2.2. Structure of Wrapped Functions

In the previous example (Figure 1), the wrapper substitutes a **C** library function with a different function. More commonly, a wrapped function performs the same functionalities as the original function but provides more error checking. Figure 3 illustrates the typical structure of a wrapped function f_w that consists of some prefix code, a call to the original function f_o, and some postfix code. The prefix code, for example, may check that the arguments passed to the function are valid. If not, it can return an error code without executing the original function. For example, the `fopen(path, mode)` function causes a segmentation fault if the `mode` flag is a null pointer. To avoid a program crash, the wrapped function returns an error code `EINVAL` (invalid argument) when it detects that the second argument is not a valid **C** string. Similarly, the postfix code can check if the function execution results in an error. For example, the `malloc` function returns a `NULL` point if the memory allocation fails. In this case, the wrapped function can write an error message in a log file for failure diagnosis or may even try to overcome the error. We will describe how the wrapper can overcome failures due to temporary resource unavailability in Section 3.

The generator emits slightly better code if the postfix code is empty. In this case the wrapper jumps to the original function instead of calling the original function. In this way, the wrapper does not need to copy the arguments of

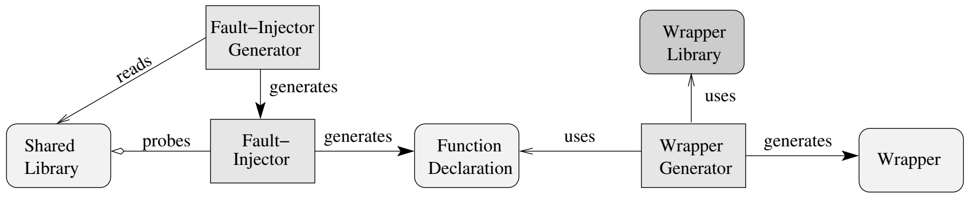


Figure 2. Architecture of the wrapper generation process.

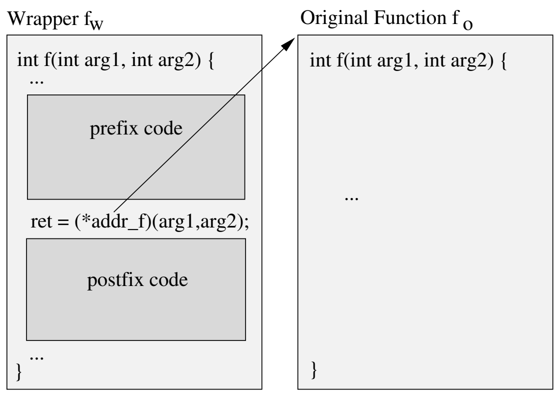

Wrapper f_W
Original Function f_O
int f(int arg1, int arg2) {
...
prefix code
ret = (*addr_f)(arg1,arg2);
postfix code
...
}
int f(int arg1, int arg2) {
...
}

Figure 3. A wrapper f_w of function f consist of some prefix code, a call to the original function f, and some postfix code. The address of the original function is stored in variable `addr_f`.

the function and can instead use the arguments provided by the call to the original function.

2.3. Function Declarations

One challenge in the design of a wrapper generator is how to efficiently generate a large number of wrapped functions. Clearly it is undesirable to write every function from scratch. Our approach is to generate function declarations by parsing header files and manual pages and the use of automated fault-injection experiments [4]. A function declaration includes (in addition to other information):

- Function name.
- Type of arguments and their valid domains.
- Return type of the function.
- Return value if an error occurs. For example, many **C** library functions like `open` return -1 on an error and an non-negative number when no error has occurred.
- Pre-,post-, and argument conditions. Some functions do not define an error return value even though they can fail. For example, `abs` (absolute value of an integer) can fail due to an arithmetic overflow: `abs` of the smallest negative number results in the smallest negative number for machines that use 2s complement to represent integers. To detect when such functions fail, we permit the definition of pre- and post-conditions in addition to constraints for arguments.
- Set of possible error codes (i.e., possible values for `errno`). This set might be empty.
- Attributes of the function (described later).

Figure 4 shows an example of a function declaration for `abs`. As previously mentioned, this function fails if its argument is the smallest negative integer (denoted by *MIN_INT*). Hence the declaration specifies that the argument must not be *MIN_INT*. The generator can emit code to check the argument. If the argument is invalid, the wrapper returns an error return value (specified by field $error\_value$) and sets `errno` to *EINVAL* (invalid argument). This directs the wrapper generator to produce the argument checking code as shown in Figure 5. The code includes a recursion detection flag `in_wrapper` to avoid potential circular dependencies that may occur during the function resolution process (see Section 4 for details). The generator can additionally emit code that logs this failure before it returns.

Function *attributes* reflect certain properties of the function that can be useful to the wrapper generator. The following is a partial list of function attributes.

- *const*: This is a mathematical function. Given the same input, it always produces the same output. It does not rely on any system state. Examples: `exp`, `floor`, `sin`.
- *query*: The result of the function depends on the state of the system. However, it does not modify any state. Examples: `feof`, `getcwd`, `getpid`.

```
<function>
        <name>abs</name>
        <argument>
                int <cond>!=MIN_INT</cond>
        </argument>
        <return_type>int</return_type>
        <error_value>MAX_INT</error_value>
        <attribute>const</attribute>
</function>
```

Figure 4. Declaration of function `abs`.

```
int abs(int arg1) {
        int ret = MAX_INT;
        if (in_wrapper) {
                return (*addr_abs)(arg1) ;
        }
        in_wrapper = true ;
        if (!(arg1 != MIN_INT)) {
                errno = EINVAL ;
        } else {
                ret = (*addr_abs)(arg1) ;
        }
        in_wrapper = false ;
        return ret ;
}
```

Figure 5. Wrapped function for `abs`.

- *update*: This function changes system state and its result may depend on some system state. Examples: `malloc`, `fopen`, `mkdir`.
- *signal*: This function causes a signal to be thrown. Examples: `alarm`, `abort`.
- *fexception*: This function can raise a floating-point exception. Examples: `div`.
- *atomic*: This function does not modify any system state if it returns an error code. Examples: `malloc`, `fopen`.
- *safe*: This function already checks that all arguments passed to it are valid. There is no need for additional argument checking in the wrapped function. Examples: `log`, `alarm`, `strerror`.
- *unsafe*: This function does not check all its arguments. The function might hang or abort if called with an invalid argument. Additional argument checking is needed in the wrapped function. Examples: `strlen`, `atof`.
- *noreturn*: This function does not return. Examples: `abort`, `exit`.

Function attributes can be useful in generating the wrapper. For example, if a function is *const*, then its correctness does not rely on any system state. In contrast, if a function is *query* or *update*, then additional state keeping may be needed. One example is the `free` function which frees a block of previously allocated memory. The wrapped `free` function needs to check that the memory to be freed was previously allocated by `malloc`, `calloc`, or `realloc`. Otherwise, it should return an error code instead of executing the function. This requires the wrapper to keep track of the memory allocation status in the system. In [3] we described how our wrapper can be used to detect buffer overflows in existing systems. The *safe* versus *unsafe* properties describe the robustness of a function with respect to invalid inputs. These properties are used in the generation of robustness wrappers as described in [4].

The *atomic* attribute is an interesting one. A function is *atomic* if it does not modify any system state upon failures. Hence one can safely *retry* this function if the failure is due to a temporary shortage in resources. We will explore this and a weaker attribute, *failure-idempotent*, in detail in Section 3.

We automatically generated function declarations for more than 800 functions in the **C** library using the technique described in [4]. The advantage of this approach is that a function declaration allows us to focus on the unique properties of the function instead of writing a lot of similarly structured code.

2.4. Wrapper Generation

After the function declarations have been generated, our system creates a set of wrappers through the wrapper generator. The challenge here is that applications may have different levels of reliability and security requirements and hence, need different wrapper support. For example, an application with root privilege usually needs a higher level of security assurance than a user application. Similarly, an application running inside a virtual private network can have very different security requirements from an application running outside the firewall. Moreover, even the same application may need different types of wrappers throughout its life-cycle: during the product development phase, a wrapper may abort the execution of an application upon detection of a robustness violation (e.g., writing to an invalid memory address). In contrast, after the application has been deployed, a robustness wrapper should try to keep the application running and log invalid arguments for a later failure diagnosis.

This raises the question as how to generate a rich set of wrappers to suit the diverse needs of applications. Writing

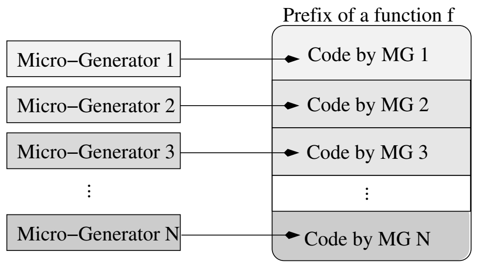


Figure 6. The prefix code consists of code fragments generated by a set of micro-generators. The order of the postfix code is reversed, i.e., code generated by MG N is first and the code of MG 1 is last.

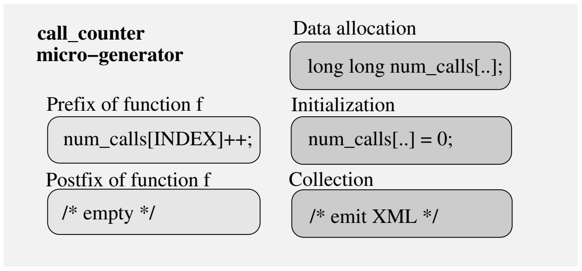


Figure 7. Structure of the call_counter micro-generator.

each wrapper type manually is time consuming and error prone. It is difficult to maintain the correctness and consistency of the wrappers. For example, a new library version might contain new functions and the robustness of old functions might have changed. Each manually written wrapper would have to be checked and potentially updated for each new library version.

We address this problem through a modular approach where the functionality of a wrapper generator is decomposed into a number of "features", each supported by a micro-generator. Each micro-generator generates a fragment of the prefix and postfix code of a function. The micro-generators can be switched on or off individually at generation time to implement the required features for a particular application. Each micro-generator has a fixed priority. The priority determines the order of the generated code in a wrapped function as illustrated in Figure 6.

Our modular generator architecture proves to be an efficient way to implement a large number of wrapper types: since each micro-generator only implements a single feature, it is easy to optimize and reuse. The wrapper generator is highly flexible and one can easily add new micro-generators to support new features, which can then be combined with existing micro-generators to implement a new wrapper type.

2.5. Micro-Generators

To add a new micro-generator, one can overload a small number of methods of a generic micro-generator (which is provided in the form of a base class). There is one method for each of the following code fragments:

- *Data Allocation Code Fragment:* defines the global variables needed by the wrapper.
- *Initialization Code Fragment:* initializes the variables defined by the data allocation fragment.
- *Prefix Code Fragment:* defined for each wrapped function f_w and is located in the prefix of f_w.
- *Postfix Code Fragment:* defined for each wrapped function f_w and is located in the postfix of f_w.
- *Collection Code Fragment:* uploads the data collected by the wrapper.

For example, Figure 7 illustrates the structure of a "call_counter" micro-generator that counts how many times a wrapped function is called. The prefix code in the generated wrapper increments a counter associated with each function. The postfix code is empty. The micro-generator also generates code for allocating the required data structures, initializing them, and collecting the results. In this example, it allocates an array of counters that contains an entry for each wrapped function. This array is initialized to zero when the wrapper is loaded. Just before the application terminates, the collection code is called to send the gathered information to a central server. Since different types of wrappers can be used in a distributed environment, the gathered information sent to the server is in form of a "self-describing" XML document. The server can extract from the document which functions were wrapped and what kind of information was collected. Such information is then stored for later processing.

Figure 8 depicts the wrapper function `wctrans`. This code is generated by combining three micro-generators[1]: *prototype*, *call_counter*, and *caller*. The *prototype* and *caller* are standard micro-generators that generate the prototype of the wrapper function and the call to the original function, respectively. Note that all code generation is done by micro-generators to increase the flexibility of the system. For example, we could disable the micro-generator *caller*

[1]Note that since this wrapper neither calls any functions in the prefix nor in the postfix code, we do not need to enable a recursion detection micro-generator (see Section 4).

```
/* Prefix code by micro-gen prototype */
wctrans_t wctrans(const char* a1) {
   wctrans_t ret;
/* Prefix code by micro-gen call_counter */
   num_calls[1206]++;
/* Postfix code by micro-gen caller */
   ret = (*addr_wctrans)(a1);
/* Postfix code by micro-gen prototype */
   return ret;
}
```

Figure 8. Wrapper for function `wctrans` generated by three micro-generators: *prototype*, *call_counter*, **and** *caller*.

and enable one that generates jumps to original function instead – which saves copying the arguments of the function.

3. Retry Wrappers

Some of the Heisenbugs, i.e., non-deterministic bugs, are caused by resource depletion, such as depletion of memory or file descriptors. Resource depletion failures can result in the erroneous termination of programs and can be very hard to reproduce and debug. One can recover from such failures by retrying failed function calls given that there is some fluctuation in the resource usage. To reduce the work for program developers and to increase the robustness of existing programs, we created a micro-generator that generates code to retry failed function calls.

A retry wrapper first needs to decide which functions are retryable. As described earlier, an *atomic* function does not change the state of the system in case it returns an error. Hence, it can be safely retried if the failure is due to transient problems in the execution environment. Determining if a function is `atomic` can be difficult. One would have to look into the source code of the function to determine its exact behavior in case an error occurs. However, even non-atomic functions might be retryable. For example, consider function `readlink` that reads the value of a symbol link and stores it in a given argument `buf`. Typically, an implementation of this function will be atomic: it will not change `buf` if it returns an error code. However, the behavior of functions in case an error occurs may not be well defined and could differ between implementations. Nevertheless, even if `readlink` would change `buf`, one could retry the function since the retry will overwrite previous changes made to `buf`. Consequently, we introduce a weaker property, *failure-idempotent*, to simplify the determination if a function is retryable.

3.1 Failure-Idempotent Functions

Most functions have to be modeled as mathematical relations since they can be non-deterministic. For example, resource depletions failures can result in non-deterministic behavior of functions. Let f be a (programming language) function that takes an argument of type A and transforms the state S of a program and returns a value of type T. We can model f as a relation, i.e.,

$$f \subseteq S \times A \times S \times T \qquad .$$

We assume that if f fails due to a resource depletion failure, it returns a value in some set $E \subset T \wedge E \neq \emptyset$. For f to be *failure-idempotent* it has to satisfy the following conditions:

1. When one retries the execution of a failure-idempotent function after a resource depletion failure, there exists the possibility that this retry succeeds: $\forall s_1, s_2 \in S, \forall a \in A, \forall e \in E : (s_1, a, s_2, e) \in f \Rightarrow \exists s_3 \in S, \exists t \in T - E : (s_2, a, s_3, t) \in f$.

2. A retry of a failure-idempotent function does not change the semantics of the function – the result of the retry must be equivalent to some single execution of f: $\forall s_1, s_2, s_3 \in S, \forall a \in A, \forall e \in E, \forall t \in T : (s_1, a, s_2, e) \in f \wedge (s_2, a, s_3, t) \in f \Rightarrow (s_1, a, s_3, t) \in f$.

The `readlink` function in the previous example is failure-idempotent. First, there exists the possibility that a retry succeeds in case it previously failed due to insufficient kernel memory (`errno` is set to `ENOMEM`). Second, a retry will overwrite any modification of argument `buf` that might have been done during previous attempts of executing `readlink`. A function declaration can specify if the function is failure-idempotent and the set of error codes that indicate a resource depletion failure. Only if the function declaration says that it is failure-idempotent or atomic will the micro-generator be permitted to generate retry code. The declaration of function `readlink` and the generated retry code are depicted in Figure 9 and 10, respectively. Note that an idempotent function is not necessarily a failure-idempotent function. However, all atomic functions are failure-idempotent (assuming they return an error code if they fail).

3.2 Resource Manager

Sometimes resource contention due to competing processes may lead to situations that cannot be easily resolved by retrying. Consider a system that consists of multiple application processes, each of which requires a certain number of file descriptors. It may happen that the total number of file descriptors required by all processes exceeds the

```
(1) <function>
(2)  <name>readlink</name>
(3)  ...
(4)  <error_return>-1</error_return>
(5)  <attribute>
(6)        failure-idempotent
(7)        <errno>ENOMEM</errno>
(8)  </attribute>
(9)  ...
(10) </function>
```

Figure 9. Declaration of function `readlink`.

```
int readlink(const char *path, char *buf, size_t bufsiz) {
  ......
  {
    bool contRetry = true; int retry = 0;
    for (retry = 0 ; retry < MAXRETRY
      && contRetry ; ++ retry) {
      contRetry = false;
      ......
      ret = readlink(path, buf, bufsiz);
      ......
      if ((ret == -1) && errno == ENOMEM) {
        contRetry = true;
        need_memory(); usleep(RETRY_DELAY);
      }
    }
  }
  ......
}
```

Figure 10. Retry code generated for `readlink`.

maximum number of file descriptors available in the system. In this case, if each process just keeps retrying, the system will not make progress since all resources are occupied.[2] Ideally, in this case certain low-priority processes should either be terminated or voluntarily free up resources for high-priority processes.

We address this problem by using a resource manager in the system. When a process starts, its wrapper reports the id and the priority of the process to the resource manager. (For simplicity, we assume there exists some external mechanism to assign a priority to a process based on its importance.) When a function call fails due to resource depletion (e.g., `fopen` returns a `NULL` pointer and sets `errno` to `ENFILE`), the wrapper notifies the resource manager of the problem. The resource manager then attempts to free some resources by sending low priority processes a signal that they should either free up file descriptors or that they have to terminate. This is illustrated in Figure 11. Voluntary release of resources requires the applications to be modified. They need to implement a signal handler that catches the signals sent by the resource manager and release resources in response to such signals. Applications that do not provide such a signal handler are terminated as soon as they receive the first signal from the resource manager.

[2]Whether such a situation occurs in practice depends on the relative timing of different processes in acquiring and releasing resources.

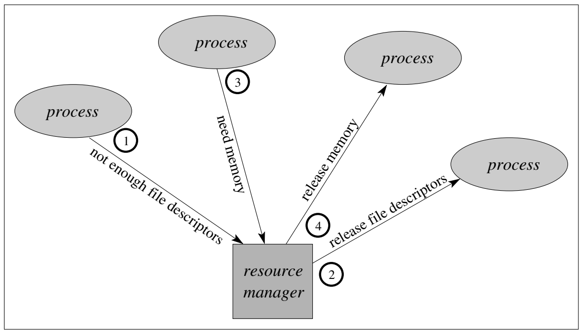
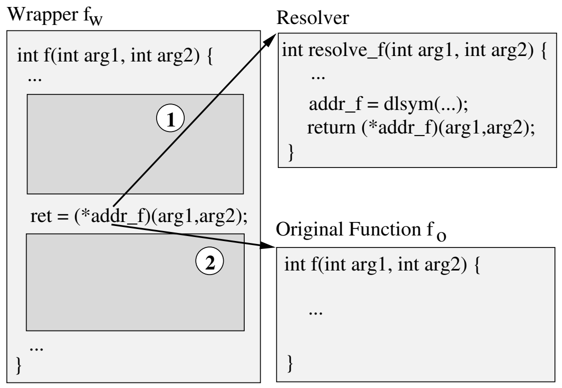


Figure 11. The resource manager can decide what a process should do in case of resource contention.

We evaluated the effectiveness of our retry wrapper in a system with 6 processes. Each process repeatedly tries to acquire 1000 file descriptors, and then releases all of them. The maximum number of file descriptors for all processes in the system is 4096. Consequently, a process occasionally may fail to obtain a file descriptor due to resource depletion. In this case, the resource manager asks one of the processes to give up all its file descriptors. We measured how many times it takes for the retry wrapper to open a file successfully. Our results indicate that 99.9% of the time a process was able to open the file in the first try. However, sometimes a process may need to retry several times before it can obtain a file descriptor. For example, 0.04% of the time it took the wrapper 21 tries to open a file (20 microsecond sleep between retries). The maximum number of retries we observed in this experiment was 61.

4. Implementation Issues

As described earlier, we increase the robustness of applications by running a wrapper that sits between an application and its shared libraries. For a wrapped function f to be able to call the original function, it needs to find out the



```
int f(int arg1, int arg2) {
  ...

  ret = (*addr_f)(arg1,arg2);

  ...
}
```

```
int resolve_f(int arg1, int arg2) {
    ...
    addr_f = dlsym(...);
    return (*addr_f)(arg1,arg2);
}
```

```
int f(int arg1, int arg2) {

    ...

}
```

Figure 12. Initially, variable `addr_f` points to a generated function `resolve_f`. Upon the first call via `addr_f`, function `resolve_f` stores the address of f in `addr_f` and then calls f_o.

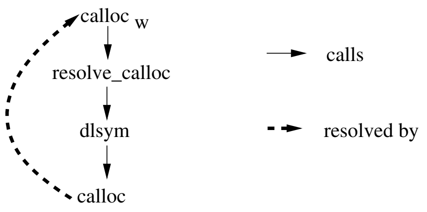
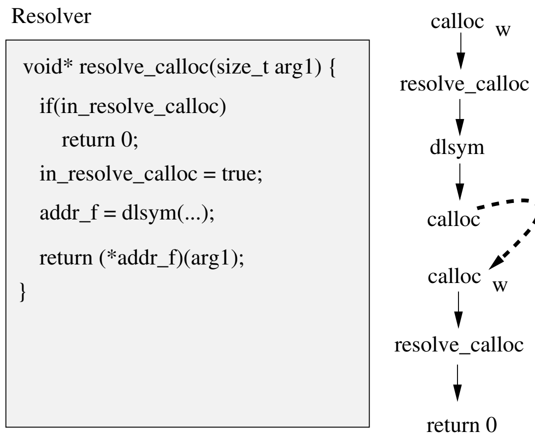


Figure 13. Resolving `calloc` can introduce circular dependencies for implementations of `dlsym` that need to call `calloc`.



```
void* resolve_calloc(size_t arg1) {
  if(in_resolve_calloc)
     return 0;
  in_resolve_calloc = true;
  addr_f = dlsym(...);
  return (*addr_f)(arg1);
}
```

Figure 14. Circular dependencies caused by `dlsym` are detected and infinite recursion avoided by resolving a symbol at most once per thread.

address of the original function. (Note that if the wrapped function just calls f, it would end up calling itself instead of the original function.) This is achieved through a resolver function as illustrated in Figure 12. Initially the variable `addr_f` contains the address of the resolver function. During the first time f is called, the resolver function finds the address of f by resolving symbol f through an interface function `dlsym` of the dynamic link loader. Then it stores the address in variable `addr_f` for later use. Hence the overhead of the resolution is only incurred during the first call to f.

We try to resolve all wrapped functions when a wrapper is initialized or when the first wrapped function is called – whichever happens first. If a wrapper cannot resolve all function names successfully, it exits before the application starts executing its main function. This provides failure atomicity.

In some situations, the `dlsym` function may issue calls to wrapped functions that have not been resolved yet. For example, while resolving symbols with the GNU C library (version 2.2), we experienced calls to the following not yet resolved functions: `calloc`, `pthread_cond_broadcast`, `pthread_mutex_unlock`, `pthread_mutex_lock`, and `pthread_cond_broadcast`. This can introduce circular dependencies as illustrated in Figure 13 for the `calloc` function. Suppose the wrapped `calloc` function calls `dlsym` to resolve the original function. In Unix (unlike in Windows), this results in another call to the wrapped `calloc` instead of to the original function. If the wrapped `calloc` again calls `dlsym` to resolve the name, it leads to an infinite recursion that ultimately ends up with a stack overflow.

We address this problem by associating a recursion detection flag with each wrapped function. Initially, all these flags are false. When a resolver function is called, it checks whether the flag is set. If not, it sets the flag and calls `dlsym` to resolve the name. Otherwise, it returns a function specific error code instead of calling `dlsym` again. (The flag is never cleared since each function needs to be resolved at most once per thread.) Figure 14 illustrates this process for the `calloc` function in the previous example. The wrapper maintains a flag `in_resolve_calloc` to record whether the resolver function `resolve_calloc` has been called previously. When the `dlsym` function calls the wrapped `calloc` function to allocate memory, the resolver function detects the recursion and returns an `NULL` pointer. The GNU implementation of `dlsym` (GNU C library version 2.2) uses in this case a statically allocated buffer for the resolution instead.

Another form of circular dependencies can occur when

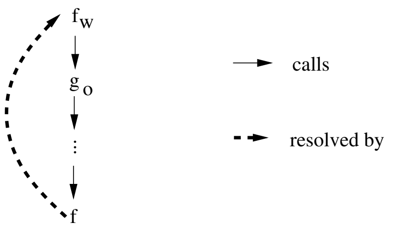
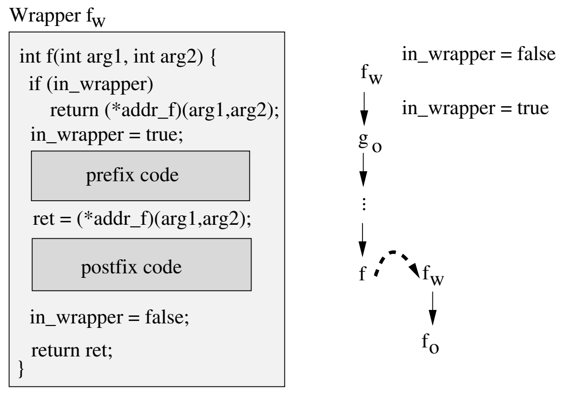


Figure 15. Circular dependencies can occur if the prefix or the postfix code of f_w calls a function g_o which in turn calls f (i.e., f_w).



```
int f(int arg1, int arg2) {
  if (in_wrapper)
      return (*addr_f)(arg1,arg2);
  in_wrapper = true;
      prefix code
  ret = (*addr_f)(arg1,arg2);
      postfix code
  in_wrapper = false;
  return ret;
}
```

Figure 16. Circular dependencies caused by prefix or postfix code are avoided by skipping the execution of these code fragments in case of a recursive call.

the prefix or postfix code of a wrapped function f calls another function g which in turn calls f. For example, the postfix code of `write` might call `printf` to print out tracing information and `printf` might call `write` to output this information on the console. Typically, we restrict function calls made inside a wrapped function to original functions only. Hence in this case the wrapped function f will invoke the original function g. However, the implementation of function g may need to call f, in which case the wrapped function f is invoked. This is illustrated in Figure 15. Such dependencies between functions are implementation specific and can vary across different library vendors or even between different versions of a library from the same vendor.

To avoid such recursions, we would like to have function g call the original function f instead of the wrapped version. This is achieved by maintaining a boolean flag `in_wrapper` for each thread. This flag indicates whether the thread is currently executing a wrapped function. Initially, all these flags are false. When a wrapped function is invoked, it checks whether the flag is set. If so, it invokes the original function directly without executing any prefix or postfix code. This effectively ensures that any function call made inside a wrapped function (either directly or indirectly) invokes its original version instead of the wrapped version.

The flag `in_wrapper` is cleared before a wrapped function returns. Note that when a wrapper is interrupted by a signal, the flag is cleared too. This can be done efficiently by intercepting calls to signal handlers. Figure 16 shows the pseudo code of a wrapped function f. The code for the recursion detection is generated by a micro-generator. This allows us to disable the recursion detection if no function is called in the prefix and the postfix code of a particular wrapper type, such as the profiling wrapper described in Section 2.5.

5. Performance

In this section, we measure the performance overhead of the generated wrappers. The measurements were conducted on a 864MHz Intel Pentium III system with 384MBytes of memory running a Linux 2.4.4 kernel and the GNU **C** library version 2.2. Each data point in the measurements is the 10% trimmed mean of 100 executions: we repeated each experiment 100 times, filtered out the 10 smallest and 10 largest values, and then computed the average. This ensures better reproducibility of the results than the more commonly used "mean".

5.1. Loading Overhead

Consider that we want to run a program with a wrapper. Before the program can start executing, the wrapper needs to be loaded into memory and the functions it defines are added to the set of symbols maintained by the dynamic link loader. Hence, preloading a wrapper incurs certain overhead. We measure how the loading overhead changes with respect to the size of the wrapper. To do so, we generated wrappers of different sizes. Each of them defines exactly one function and uses padding of arbitrary bytes to achieve the required size. We also generate a test program that contains one undefined function. We measure the difference in execution time of the test program with and without a wrapper. This difference is mostly due to the loading overhead of the wrapper. Figure 17 indicates that the overhead increases only slowly with the wrapper size. Recall that we take the 10% trimmed mean of 100 executions. Hence, the small overhead is most likely because the wrapper is already in memory or cache most of the time. Since multiple programs

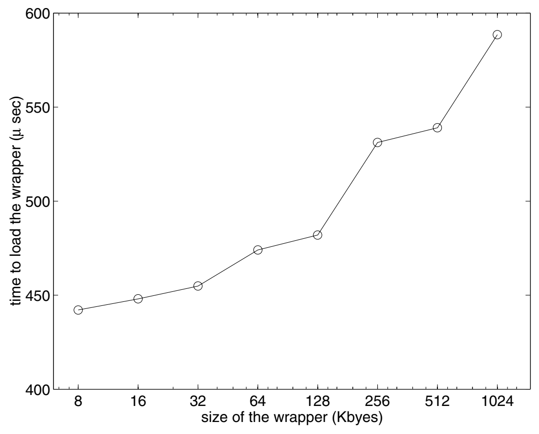


Figure 17. Overhead of loading a wrapper with respect to its size. The x-axis is in log scale.

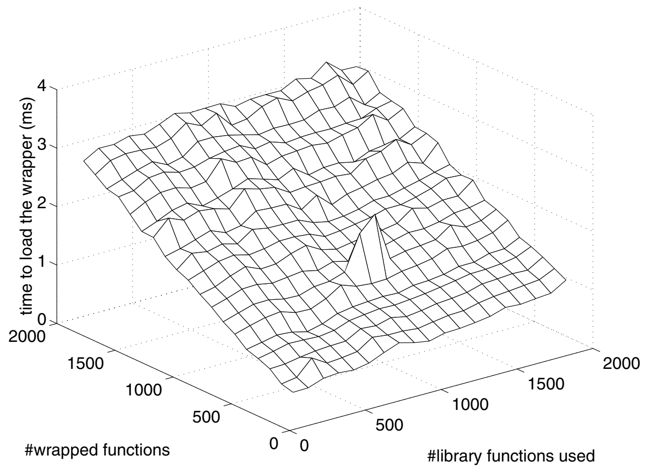


Figure 18. Overhead of linking increases with the number of functions defined by the wrapper and the number of functions used by a program.

may use the same wrapper in practice, we think this is a realistic measurement of the actual overhead. Note that even if the wrapper is already in memory, it has to be mapped into the address space of the starting process. Hence, the loading overhead grows with the size of the wrapper and is not constant.

5.2. Linking Overhead

After the wrapper and the program are loaded into memory, the dynamic link loader needs to link undefined symbols used in the program with the functions defined in the wrapper or the **C** library. The linking overhead depends on two factors: the number of functions defined by the wrapper and the number of functions used by an application (i.e, undefined symbols). In the second experiment, we generate wrappers that define between 100 and 1900 functions (in increments of 100). For simplicity, these functions are different from the existing functions defined in the GNU **C** library, i.e., we do not overload any **C** library function. We also generate test programs that use different subsets of these functions. We measure the overhead in execution time and show the results in Figure 18. As can be seen from the figure, the more functions used in a test program, the higher the overhead. This is because the dynamic link loader has to resolve these functions during startup. The overhead also grows with the number of functions defined by the wrapper. This is because such functions are added to the set of symbols maintained by the dynamic link loader and the overhead of resolving undefined symbols increases with the total number of defined symbols.

5.3. Resolution Time

After the wrapper and the program are loaded and linked, the wrapper starts executing. During start-up the wrapper needs to resolve each wrapped function f_w using the dynamic link loader API function `dlsym`, i.e., it has to find the original function f_o (see Section 2.2).

To determine the overhead due to resolution of wrapped functions, we generated wrappers that define between 100 and 2000 functions. We also generated a test program that calls `dlsym` to resolve a randomly selected function name defined in the wrapper. The results are shown in Figure 19. The figure indicates that the resolution time increases with the number of functions defined in the wrapper.

6. Related Work

Software wrappers were previously used in the Xept project to handle exceptional conditions in the **C** library [13]. Xept provides a language to write exception specifications for certain **C** functions as well as a convenient framework to incorporate such specifications into application code. For function calls that failed due to temporary resource shortage in the system, Xept can generate retry code to mask the exceptions from application programs. Like Xept, our system can generate retry wrappers to handle transient problems in the execution environment. Moreover, it utilizes a resource manager to arbitrate resource contention among competing processes based on their priorities.

An advantage of our system is that it provides a flexible

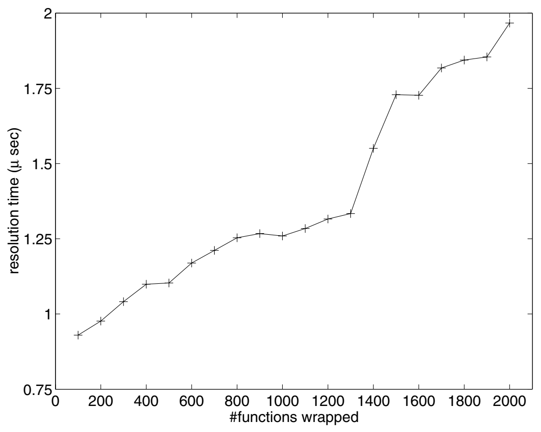


Figure 19. The time needed to resolve a symbol using `dlsym` increases with the number of functions defined in the wrapper.

trade-off between reliability and efficiency. High-level reliability requirements in the system are decomposed into a set of orthogonal properties that are implemented by micro-generators. This allows a new wrapper type to be created easily by composing a set of micro-generators to suit the needs of a specific application. Such a modular architecture was built on previous work in micro-kernel designs and group communication systems. x-Kernel, for example, defines a modular structure for implementing network software [9].

The authors of [10] used fault containment wrappers to improve the robustness of COTS micro-kernels. By verifying certain predicates when a system call is performed, the wrapper detects errors due to corrupted parameters and may optionally perform some corrective actions to restore the system into a consistent state. Horus [12] and Ensemble [6] are two examples of modular group communication systems where protocol layers can be stacked on top of each other in a variety of ways.

Exception handling mechanisms in several languages have been studied in [2]. Some of them focus on language design. For example, both Mesa [8] and exceptional **C** [5] provide explicit support for retry semantics. Languages like Java [1] provide garbage collection support and eliminate a large class of memory related errors. In contrast, our focus is on providing transparent support for existing **C** programs even if the source code is not available. We believe the two approaches are complementary to each other and both represent important research directions to pursue.

7. Conclusion

Wrapping dynamic link libraries is an effective approach for improving the reliability and security of critical applications without source code access. This paper describes a flexible framework to generate a rich set of software wrappers through the concept of micro-generators. Given a declarative description of the functions in a library, our system can customize the generated wrapper types to fit the diverse requirements of application programs. The wrapper generator also provides support to address certain implementation issues we encountered during the development process, such as detecting circular dependencies among wrapped functions to avoid unbounded recursions. Extensive performance measurements demonstrate that the overhead of generated wrappers is small.

Acknowledgments

We like to thank Bob Gruber and the anonymous reviewers for comments on an early draft of the paper.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, June 2000.

[2] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[3] Christof Fetzer and Zhen Xiao. Detecting heap smashing attacks through fault containment wrappers. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, October 2001.

[4] Christof Fetzer and Zhen Xiao. An automated approach to increasing the robustness of C libraries. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[5] N. H. Gehani. Exceptional C or C with Exceptions. *Software Practice and Experience*, 22(10):827–848, October 1992.

[6] Mark Hayden. *The Ensemble System*. PhD thesis, January 1998.

[7] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2000.

[8] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual, April 1979.

[9] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[10] Frederic Salles, Manuel Rodriguez, Jean-Charles Fabre, and Jean Arlat. Metakernels and fault containment wrappers. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[11] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[12] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. In *Communications of the ACM*, April 1996.

[13] K-P. Vo, Y-M. Wang, P. Chung, and Y. Huang. Xept: a software instrumentation method for exception handling. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 60–69, Albuquerque, NM, USA, Nov 1997.