# An Automated Approach to Increasing the Robustness of C Libraries

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

## Abstract

*As our reliance on computers increases, so does the need for robust software. Previous studies have shown that many* **C** *libraries exhibit robustness problems due to exceptional inputs. This paper describes the HEALERS system that uses an automated approach to increasing the robustness of* **C** *libraries without source code access. The system extracts the* **C** *type information for a shared library using header files and manual pages. Then it generates for each global function a fault-injector to determine a "robust" argument type for each argument. Based on this information and optionally, some manual editing, the system generates a robustness wrapper that performs careful argument checking before invoking* **C** *library functions. A robustness evaluation using Ballista tests has shown that our wrapper can prevent crash, hang, and abort failures. Moreover, the wrapper generation process is highly automated and can easily adapt to new library releases.*

## 1. Introduction

Traditionally, computer software has been optimized for efficiency with robustness only as a secondary consideration. During the past several decades, we have seen an exponential increase in processing power. Consequently, one might expect that efficiency is becoming less of an issue and that more systems will be developed using safe languages like Java or ML. In practice, however, the demand for computing power in emerging applications (e.g., data mining) has also increased rapidly. Therefore, there is still a strong demand for efficient programming.

**C** and **C++** are known as two of the most efficient languages. In addition, **C** and **C++** give programmers extensive control over system resources, in particular, memory. Compared with languages that support automatic memory management (e.g., garbage collection), **C** and **C++** permit savvy programmers to optimize resource usage based on application knowledge. This explicit control over memory also supports memory mapped I/O, which is important for system-level programming. In fact, for safe languages it is often necessary to introduce extra functionality through customized **C**-stubs.

A large percentage of existing software is written in efficient but unsafe languages like **C** or **C++**. Most **C** libraries are optimized for high performance at the expenses of system robustness. The Ballista team [8, 6] has shown that many POSIX **C** library functions are brittle with respect to invalid inputs: a function invoked with invalid inputs may cause the calling process to crash, hang, or give erroneous results. This is because many functions make implicit assumptions about their arguments and often omit validity checks for efficiency reasons. For example, the `strcpy(dst, src)` function assumes that the location pointed to by `dst` is writable and has enough space to accommodate the source string. Violating this assumption may cause a segmentation fault or a security vulnerability. Such a design prevents correct programs from being penalized by unnecessary checks.

With the increasing reliance on computers, there is a vital need for tools that increase the robustness of software. The emphasis of **C** and **C++** on efficiency and explicit resource control makes it difficult to build robust and secure libraries. There are some safe libraries that increase the robustness of a subset of the standard **C** library without a substantial performance overhead [7]. In this paper, we describe a system called *HEALERS* (HEALers Enhanced Robustness and Security) that does not attempt to modify the library itself. Rather, it uses an automated approach to generate robustness wrappers based on adaptive fault-injection experiments. A fault-injector for a given library function $f$ calculates a *robust* argument type for each argument of $f$: if $f$ is called with an argument that is not an element of the computed robust argument type, $f$ will crash or return with an error. A robustness wrapper checks that each argument passed to a function belongs to the specified robust

Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)
0-7695-1597-5/02 $17.00 © 2002 IEEE

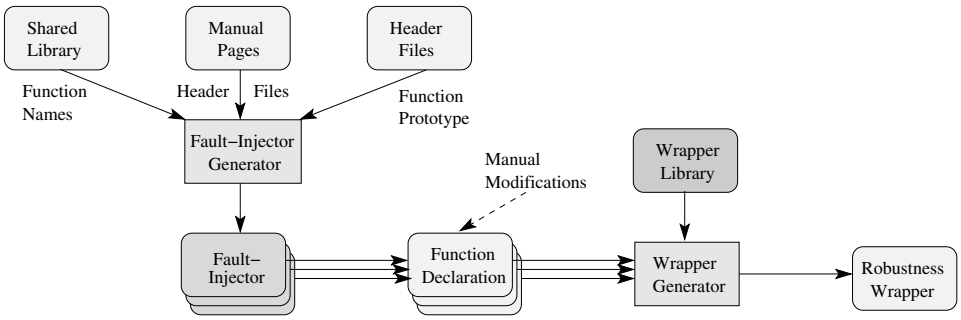


**Figure 1. Architecture of the wrapper generation process.**

argument type before invoking the function. Otherwise, the wrapper returns an error code indicating that the function was called with an invalid argument. The advantage of this approach is that one can improve the robustness and the security of libraries that are only available as binaries.

The rest of the paper is organized as follows. We first introduce the architecture of the HEALERS system in Section 2. Then, in Sections 3 and 4, we show how we use automated fault-injection experiments to determine what kind of argument checks a robustness wrapper has to perform for individual functions. Section 5 describes how HEALERS creates robustness wrappers and some techniques that we used for argument checking. We evaluate the robustness and performance of our wrappers in Sections 6 and 7. Related work is described in Section 8. Section 9 concludes the paper.

## 2. Architecture

We increase the robustness of applications by running a wrapper that sits between an application and its shared libraries. The wrapper itself is a shared library that has priority over the other libraries to resolve undefined symbols of the application. Figure 1 depicts the wrapper generation process that consists of two phases. In the first phase, our system extracts the **C** type information for the global functions in a shared library using header files and manual pages. Based on this information, it generates for each global function a fault-injector to compute a *robust* argument type for each argument of the function. The fault-injectors generate function declarations that are fed into the wrapper generator. Some function declarations may need manual editing to prevent all robustness violations. However, this task is largely automated.

In the second phase, the wrapper generator can generate a variety of wrappers to suit the needs of individual applications. In this paper, we focus on how to generate robustness wrappers that prevent all crash and hang failures discovered by the Ballista test suite. Robustness wrappers in our system provide a flexible trade-off between efficiency and robustness. If needed, a system developer could decide which functions should be wrapped and which processes should run with a wrapper. For example, a process with root privilege may use our wrapper to detect buffer overflow attacks [4] that are a major cause of security breaches in modern operating systems. In contrast, a process owned by an ordinary user may use only a minimal wrapper to prevent system crashes without much performance overhead. Moreover, different wrappers can be used in the life-cycle of an application. For example, a wrapper in the debugging phase may abort the execution of an application upon detection of an invalid input. After the application has been deployed, a wrapper should try to keep the application running and log invalid inputs.

Our wrapper-based approach can protect existing applications transparently without having to modify or recompile the applications or the **C** libraries. In addition, the wrapper generation process is highly automated and can easily adapt to new library versions. As shown in [6], new library releases are sometimes more robust than previous versions due to bug fixes, and sometimes less robust due to bugs introduced in new features. Using an automated approach greatly simplifies what would otherwise be a labor intensive and error prone process of hardening each new release of a library.

## 3. Function Declaration Generation

A function declaration describes certain properties of the function that are needed by the wrapper generator. In this section, we first describe the structure of function declarations and then explain how we derive such declarations in an automated fashion. A function description includes (in addition to other information like pre- and post-conditions):

- **Function Name**: The name and version of the function. Since the interface and the semantics of functions may evolve over time, modern libraries assign a version number to each function they define (e.g. see [9]). This allows the dynamic link loader to resolve a symbol using the correct version of the function.
- **Function Type**: The **C** type of all arguments and return value of the function.
- **Robust Argument Types**: Enforcing an argument to be of the type specified in the prototype of a function is usually not sufficient to prevent robustness failures. A *robust* argument type prevents some if not all function crash failures (see Figure 2 for an example and Section 4 for a precise definition).
- **Error Return Code**: **C** functions typically return a unique error code (e.g. $-1$) to indicate that an error has occurred. Many functions also set variable `errno` to specify the reason of the error. (`errno` is usually implemented as a function to make it thread-safe.)
- **Function Attributes**: Two attributes are used for the generation of robustness wrappers: *safe* or *unsafe*. A *safe* function already checks that all arguments passed to it are valid. There is no need for additional argument checking in the wrapper. In contrast, an *unsafe* function may crash or hang when invoked with invalid argument values and hence needs protection in the wrapper.

Figure 2 depicts the function declaration for `asctime`. Although its argument prototype in the **C** library is `const struct tm*`, its robust argument type determined by our fault-injector is either a null pointer or a pointer to an allocated and readable block of memory with at least $44$ bytes. In the remainder of this section, we explain how to extract each item in a function declaration automatically from **C** libraries.

### 3.1. Extracting Function Names

We use the utility program `objdump` to extract the name and version of all global functions defined in a shared library. Most **C** libraries adopt the convention that all function names starting with an underscore (e.g. `_IO_fflush`) denote internal functions that should not be used by applications. Hence, typically it is sufficient to only wrap functions whose names do not start with an underscore[1]. This can lead to a substantial reduction in the number of functions that have to be wrapped. For example, in *glibc2.2* more than $34\%$ of the global functions are internal. In the following, we focus only on global functions that are external.

[1] Note that some functions used by programmers (e.g., `setjmp`) might be macros that are mapped to an internal function. To avoid this problem, one could extract all undefined functions from an application instead and wrap all functions that are resolved by the library.

```
<function>
 <name>asctime</name>
 <argument>const struct tm*
  <robust_type>R_ARRAY_NULL[44]</robust_type>
 </argument>
 <return_type>char*</return_type>
 <error_value>NULL</error_value>
 <errors>
   <errno>EINVAL</errno>
 </errors>
 <attribute>unsafe</attribute>
</function>
```

**Figure 2. Function declaration for `asctime`.**

### 3.2. Extracting Function Types

Extracting function type information for functions in **C** libraries is non-trivial because such information is not stored in the shared libraries. This is different from **C++** libraries where both the name and the type of a function are encoded in its symbol name. This encoding was introduced to permit function overloading across multiple object files. It also allows easy extraction of function type information.

We address this problem in **C** libraries by parsing header files that contain the prototypes of global functions. However, it turns out that there typically does not exist a well-defined set of header files that describe the interface of a shared library. In addition, some functions are defined multiple times in different header files while the definitions of other functions are spread across multiple header files. To determine the proper set of header files that contain the full definition of a function type, we parse the manual page that describes the function. By convention, manual pages contain a list of all header files that need to be included by a program that wants to use the function.

We have experienced a few problems with this approach. The first problem is that many global functions have no manual page. For example, we found that only $51.1\%$ of the *glibc2.2* functions in SuSE LINUX 7.2 Professional are listed in its online manual. In addition, a small percentage ($1.2\%$) of manual pages do not list the header files that need to be included. Worse yet, $7.7\%$ of the manual pages list the wrong header files: none of the listed header files (nor any files included by them) define a prototype for the function. We nevertheless use the manual pages first because we have a higher chance of success in case the function is defined across multiple header files. If a function has no manual

page or its manual page does not include the proper header files, we search through all header files (below a given path) to locate the prototype of the function. Using this approach we were able to find header files for 96.0% of the *glibc2.2* functions. Note that if a function is not found in any header file, it most likely means that the function is only intended for internal use or that the function is deprecated.

After locating the header files for a function, we parse them to extract the prototype of the function. Currently this is achieved using the CINT C/C++ interpreter [5]. The advantage of CINT is that it provides an easy interface to query extended run-time type information of all functions that are declared.

### 3.3. Determining Error Return Code

When a robustness wrapper detects a robustness violation, it needs to set `errno` and then returns an error return code to notify the application of the error. We determine the error return code of a function using adaptive fault-injection experiments. Based on the extracted function name and type information described previously, our system creates a fault-injector for each function in a library. A fault-injector generates a sequence of test cases based on the argument types of a function, similar to Ballista tests. However, our goal here is different: while Ballista tests aim to detect robustness violations in a **C** function, our goal is to discover the robust argument types of a function – the types that prevent the function from crashing. A fault-injector iterates over a sequence of test cases to determine which of them results in a crash. To reduce the number of test cases, we use an adaptive testing technique where the future test cases depend on the results of previous tests (see Section 4 for details). If a test case does not result in a crash, we record the return code and the status of `errno`. We use this information to classify functions as follows:

- **No Return Code:** The function has no return value (i.e., return type is `void`) or returns a type for which neither the equal operator (==) nor the not-equal operator (`!=`) is defined. The latter is a reasonable assumption since programs typically use equal or not-equal operator to check for an error.
- **Consistent Error Return Code:** The function sets `errno` at least once during the fault-injection experiments and always returns the same value every time `errno` is set.
- **Inconsistent Error Return Code:** The function returns different values when setting `errno`.
- **No Error Return Code Found:** The function did not set `errno` during the fault-injection experiments.

### 3.4. Determining Function Attributes

In addition to determining the error return code of a function, a fault-injector also determines if the function is safe or unsafe. A function is unsafe if at least one test case causes the function to crash or to hang for some predefined time-out period. The wrapper generator creates robustness wrappers only for unsafe functions. In this way, it avoids the overhead of unnecessary argument checks.

## 4. Determining Robust Argument Types

This section describes how to determine robust argument types for a function using adaptive fault-injection experiments.

### 4.1. Fault-Injector Generation

Our system generates for each function a specialized fault-injection program. Such a fault-injector calls the function under test with a sequence of test cases. The sequence of test cases is the cross product of the test cases for each individual argument. The test case generation is adaptive in the sense that if a function crashes, our fault-injector attempts to determine which argument caused the crash based on the address where the segmentation fault occurred. It then tries a finite number of times to change that input value until the violation disappears or another argument causes the violation. For example, in order to determine the size of an array, we first allocate an array of zero size. We use hardware memory protection to make sure that an access to an element after the last allocated element generates a memory segmentation fault. By catching segmentation faults, we can determine by how much the array has to be enlarged. The array is iteratively enlarged until no more segmentation faults occur (or, we run out of memory).

The fault-injector generator (see Figure 1) uses the **C** argument type to select at least one *test case generator* for each argument of a function. A test case generator produces a finite sequence of test cases. Each of these test cases has the same **C** type. To be able to use the generator for an argument, the **C** type has to be castable to the **C** type of that argument.

Our system has generic test case generators for all basic types, pointers, and structures. This permits the generation of fault-injectors for all functions. However, we also permit the addition of new test case generators that contain specific test cases for certain types. For example, we have a generic pointer test case generator but also a specific generator for pointers to the `FILE` structure.

The generated fault-injector iterates over all test cases and when it is done, it computes for each argument a robust

argument type. The fault-injector itself is robust. To perform a call of the tested function, the fault-injector spawns a child process. The child sets a signal handler for segmentation faults and then calls the function. In most cases, the signal handler of the child will intercept segmentation faults. However, some segmentation faults cannot be intercepted (e.g., some caused by `longjmp`). This is why a child process executes the actual calls.

When a function causes a segmentation fault, the child process queries the test case generators that provided the arguments of the function if the address at which the fault occurred belongs to the test case generator. For at most one of the generators this test will be true. If one is found, that generator is called to adjust its test case and if it is able to adjust the test case, a new child process retries the function call. Otherwise, the testing continues with the next test pattern. This adaptive testing behavior is particularly useful to determine the exact amount of memory needed for an argument without using a massive number of static test cases.

### 4.2. Test Case Generators

Test case generators are essential to finding robust argument types for a function. Because **C** types are not sufficiently powerful to be used to compute robust argument types, we designed and implemented an extensible type system. Each test case generator can define a set of types and their relationship to each other and potentially to types defined by other generators. Because some of the details are important, we are a little more formal in this subsection.

We use $(\mathcal{T}, \prec)$ to refer to the partially ordered set of types defined by the test case generators. Each element $T$ of the set $\mathcal{T}$ is a type and defines a set of values $V(T) \neq \emptyset$. Relation $\prec$ defines the subtype/supertype relation over $\mathcal{T}$. A type $T_1$ is a subtype of type $T_2$ if and only if the value set $V(T_1)$ is a strict subset of $V(T_2)$, i.e., $T_1 \prec T_2 \leftrightarrow V(T_1) \subset V(T_2)$. Type $T_2$ is called a *supertype* of $T_1$. We say that a supertype is *weaker* than any of its subtypes because its value set contains more elements. Similarly, we say that a subtype is *stronger* than any of its supertypes.

There are two kinds of types in $\mathcal{T}$: *fundamental types* and *unified types*. The intuition is that each test case generator produces elements of fundamental types. The wrapper library provides for each unified type $T$ (but not necessarily for each fundamental type) a checking function that tests if a value belongs to its value set $V(T)$. The value set of a unified type $T_U$ is the union of value set of all its strict subtypes, i.e., $V(T_U) = \bigcup V(T_s) : T_s \prec T_U$. We require that the value sets of any two fundamental types $T_1$ and $T_2$ be non-overlapping: $V(T_1) \bigcap V(T_2) \neq \emptyset \Rightarrow T_1 = T_2$. A fundamental type is never a supertype.

On a practical note, in order to define unified types that have overlapping value sets, we need to define appropriate fundamental types first. For example, if we want to define the unified types of non-negative and non-positive numbers (which are overlapping), we would define three fundamental types: negative numbers, positive numbers, and zero. The non-negative numbers are represented by a unified type of the two fundamental types of positive numbers and zero.

The separation into non-overlapping fundamental types is an essential property of our algorithm to compute robust argument types. For the sake of argument, consider a 1-ary function $f$ that does not crash for non-negative arguments. If we had only non-negative and non-positive types, we would detect that $f$ crashes for some non-positive values but not all (i.e., not for value zero). Hence, we could not conclude that the robustness wrapper can check that an argument is non-negative before calling $f$ since some non-positive numbers do not crash the function too. By splitting the types in disjoined fundamental types we can compute that the non-positive test case that does not crash $f$ actually also belongs to the non-negative numbers. Hence, it is safe to check that numbers are non-negative.

Each test case generator $G$ defines a finite sequence of test cases $C_G = (c_i)_{i \in I_G}$. Note that even though the test case generation might be adaptive, a posteriori we know the sequence. While all test cases generated by a test case generator have the same **C** type, we associate each test case with a type that is used in the computation of robust argument types. Each test case $c_i$ is therefore represented by a pair $(v, T)$ such that $T \in \mathcal{T}$ is a fundamental type and $v \in V(T)$. Note that for unified types there exist no test cases.

**Test Case Generator for Fixed Size Arrays** To be more concrete, we describe a test case generator for fixed size arrays and for file pointers. The tasks of the fixed size array test case generator is (1) to determine if a pointer points to a fixed size array of a certain size, and (2) to test whether the array has to be readable and/or writable. To do this, the generator defines five fundamental types and seven unified types. The type hierarchy of fixed size arrays is depicted in Figure 3.

Each of the value sets of the three fundamental types RONLY_FIXED[s], RW_FIXED[s], and WONLY_FIXED[s] consists of a set of pointers that point to a memory region of exactly $s$ bytes and the region is either read only, readable and writable, or write only, respectively. Fundamental type NULL consists of a null pointer which points to an inaccessible memory region. Fundamental type INVALID consists of non-null pointers that each point to an inaccessible memory region.

The unified types R_ARRAY[s], RW_ARRAY[s], and W_ARRAY[s] each consists of a set of pointers that point to a memory region of *at least* $s$ bytes that is either readable, readable and writeable, or writable, respectively. These three

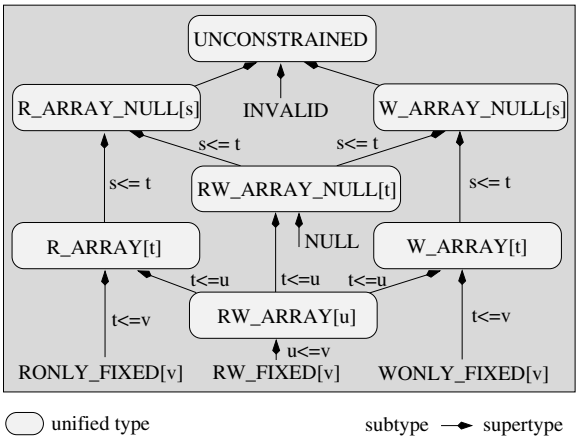


**Figure 3. Type hierarchy of fixed size arrays.**

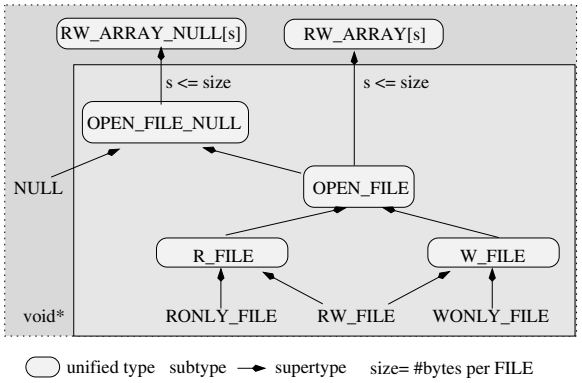


**Figure 4. Type hierarchy of file pointer types.**

unified types are combined with fundamental type NULL to define three further unified types. Finally, unified type UNCONSTRAINED is the set of all pointers.

**Test Case Generator for File Pointers** The file pointer test case generator (Figure 4) defines three fundamental types and four unified types. The value set of the fundamental types RONLY_FILE, RW_FILE, WONLY_FILE consist each of pointers to `FILE` structures that are opened for read only, read and write, or write only access, respectively. The unified types R_FILE and W_FILE are file pointers to readable or writeable files, respectively. Note that types R_FILE and W_FILE are not comparable because the intersection of their value sets is a strict non-empty subset of both their value sets. This intersection is the value set of type RW_FILE. Note however that the value set of a type can be a strict subset of the intersection of the value set of its supertypes.

Unified type OPEN_FILE is the set of all file pointers that are opened for any access pattern while OPEN_FILE_NULL is further unified with the NULL pointer type. One can define a relation between existing types and newly defined types, e.g., OPEN_FILE is a subtype of RW_ARRAY[s] (where $s \leq size$ and $size$ is the allocated size of a FILE structure). Note that the extension of an existing type hierarchy might require that previous fundamental types be redefined (or, alternatively be replaced by a new unified type). Due to the definition of the file pointer hierarchy, we need to restrict the value set of RW_FIXED[size] to make sure that the value set of OPEN_FILE and RW_FIXED[size] do not overlap.

### 4.3 Robust Argument Type Selection

We first explain how to compute robust argument types for functions with a single argument and then generalize the technique for functions with multiple arguments.

**Single Argument** Function `asctime` has one argument of **C** type `const struct tm*`. Our system uses the test case generator for fixed size arrays to inject faults into this function. The test case generator always starts with an empty array and then gradually increases its size whenever the function crashes because the size is too small.

After all test cases are performed, the fault-injector tries to compute a robust argument type $T$. Ideally, $T$ is chosen such that none of the test cases in the value set $V(T)$ causes the function to crash and any test case that is not in $V(T)$ causes the function to crash. If such a type $T$ exists, we call it the *safe* argument type. Whenever such a safe argument type exists, the robust argument type computed by our system is safe.

In the example of `asctime`, the test cases for fundamental types RONLY_FIXED[$s \geq 44$], RW_FIXED[$s \geq 44$], and NULL succeed while all other test cases fail. Figure 3 indicates that the weakest supertype of these three fundamental types is R_ARRAY_NULL[44], which actually is the safe argument type.

Sometimes such a safe type may not exist in the type hierarchy. For example, consider an implementation of `asctime` that returns an error code when invoked with pointer $-1$ instead of crashing. In this case, R_ARRAY_NULL[44] is not a safe argument type since it does not include $-1$. If we can assume that functions are atomic (i.e., a function does not change any system state if it returns an error code), then a robustness wrapper could just return an error code for all invalid pointers instead of calling `asctime` with $-1$.

We generalize the notion of safe argument type as follows. The system finds a weakest type $T$ such that all test cases for which the function returns successfully (i.e., without an error) are in $V(T)$ and for each supertype $ST$ of $T$ there exists at least one test case in $V(ST)$ for which the function crashes. Note that $V(T)$ might contain values for which the function crashes and hence, we call such an argument type *robust* instead of safe. Such a robust type always exists in our type hierarchy (due to the existence of uncon-

strained types). Note that each *safe* argument type is robust. Whenever there exists a safe argument type, the robust argument type computed by our system is safe.

We have not experienced any problems by assuming functions to be atomic. If we want to be more conservative, we can require a robust argument type to be the weakest type $T$ such that all test cases for which the function returns (with or without an error) are in $V(T)$ and as before, for each supertype $ST$ of $T$ there exists at least one test case in $V(ST)$ for which the function crashes.

**Multiple Arguments** For $n$-ary functions ($n \geq 1$) the computation of robust argument types can be generalized as follows. We need to define n-dimensional type vectors such that the $i$-th element of a vector is the type of the $i$-th argument of the function. The partial order over types defines a partial order over the type vectors. In particular, it introduces the notion of subtypes and supertypes for type vectors. Each function call by the fault-injector consists of $n$ test cases (we call this a test case vector) and these uniquely define a type vector consisting of fundamental types. The value set $V(TV)$ of a type vector $TV$ consists of a set of value vectors that is uniquely defined by the value set of the $n$ types.

We can generalize the definitions of safe type and robust type for type vectors as follows. The *safe* type vector of a function is the type vector $TV$ such that all test case vectors for which the function does not crash are in $V(TV)$ and none of the test case vectors for which the function crashes is in $V(TV)$. We call the $i$-th element of $TV$ the safe type of argument $i$. Similarly, we define the *robust* vector type of a function to be the weakest type vector $TV$ such that all test case vectors for which the function returns without an error are in $V(TV)$ and for each supertype $ST$ of $TV$ there exists at least one test case vector in $V(ST)$ for which the function crashes. We call the $i$-th element of $TV$ the robust type of argument $i$. If there exists a safe type vector, the robust type vector computed by our system is safe.

## 5. Robustness Wrapper

After the function declarations have been generated, the wrapper generator creates a robustness wrapper that substitutes each unsafe function in the declarations with a *safe* version that provides the same functionality but performs careful error checking. The typical structure of a wrapped function consists of some prefix code, a call to the original function, and some postfix code. The prefix code may check that the arguments passed to the function are "robust". If not, it returns an error return code and sets `errno` appropriately without executing the original function. The wrapper generator can additionally emit code that logs this error for a later failure diagnosis. Similarly, the postfix code can check if the function execution is successful. Figure 5 shows the wrapper code generated for function `asctime`. The prototype of the function is generated from the function declarations in Figure 2. The variable `libc_asctime` stores the address of the original `asctime` function. This code includes a recursion detection flag `in_flag` to avoid potential circular dependencies that may occur during the function resolution process. In the remainder of this section, we describe some techniques that we used to check the validity of function arguments.

```
char* asctime (const struct tm* a1) {
  char* ret;
  if (in_flag) {
    return (*libc_asctime)(a1);
  }
  in_flag = 1 ;
  if (!check_R_ARRAY_NULL(a1,44)) {
    errno = EINVAL ;
    ret = (char*) NULL;
    goto PostProcessing;
  }
  ret = (*libc_asctime)(a1) ;
  PostProcessing:  ;
  in_flag = 0 ;
  return ret;
}
```

**Figure 5. Wrapper code generated for `asctime` from the function declaration in Figure 2.**

### 5.1 Validating Memory

Many robustness violations are due to memory access errors that arise during **C** function calls. For example, functions in the *string* library often omit boundary checks of destination buffers. This has been exploited by malicious users to launch buffer overflow attacks that are a major cause of security breaches in modern operating systems.

An important feature of our system is that it keeps track of memory allocation status on the heap. When a program calls `malloc` to allocate a block of memory, the wrapper intercepts the call and records the address and size of the allocated block in an internal table. Later when the program calls a **C** library function to write to a buffer on the heap, the wrapper consults its table to locate the memory block that contains the buffer and performs boundary checks before invoking the original function. This is called *stateful* checking because the wrapper needs to perform state keeping for allocated blocks. In [4] we have shown that this technique can detect and prevent heap buffer overflows successfully.

Moreover, our wrapper can prevent stack-smashing attacks using the same mechanism as Libsafe [1].

If a buffer is neither on the heap nor on the stack, the wrapper sets up a signal handler and tests the accessibility of the memory. For large buffers that spread across multiple memory pages, only one byte per page needs to be tested. This approach was previously used in [2] to harden I/O libraries. It is a *stateless* approach because no extra state information is maintained.

### 5.2 Validating Data Structure

While validating memory accessibility can prevent most robustness violations in the string library, it is insufficient for some other libraries that may use special data structures. For example, many functions in I/O libraries use `FILE *` pointers. If an argument is a file pointer, the wrapper first needs to make sure it points to a memory region of sufficient size that is both readable and writable. Then it needs to verify that the content of the memory region correspond to a valid file structure. To do so, the wrapper calls `fileno` to extract the file descriptor in the `FILE` structure and then calls `fstat` to check the validity of the file descriptor. In theory, this is not a complete test because a `FILE` structure may be corrupted even if it happens to contain a valid file descriptor. In practice, however, we found it sufficient for almost all cases. An advantage of this approach is that the wrapper does not maintain extra state information to keep track of file pointers.

Unfortunately, **C** libraries do not provide checking functions for all its data types. For example, `closedir` expects its argument to be a `DIR *` pointer that is returned by a previous call to `opendir`. However, POSIX does not define any function to verify that a pointer points to a valid directory structure. We address this problem using a stateful approach similar to the memory tracking technique discussed previously. More specifically, the wrapper intercepts all directory related function calls and keeps track of all directory pointers in an internal table. When a program calls `closedir`, the wrapper consults its internal table to determine whether the directory pointer is valid. Stateful checking has the disadvantage that one has to switch on wrappers for a potentially larger set of functions in order to maintain state information, even though some of these functions may be safe and do not need to be wrapped otherwise.

## 6. Robustness Evaluation

In this section, we evaluate the effectiveness of the generated wrapper in preventing robustness failures. Instead of testing our wrapper on all functions in *glibc2.2*, we concentrate on the 86 POSIX functions that were previously found to suffer crash failures in the Ballista test under Linux

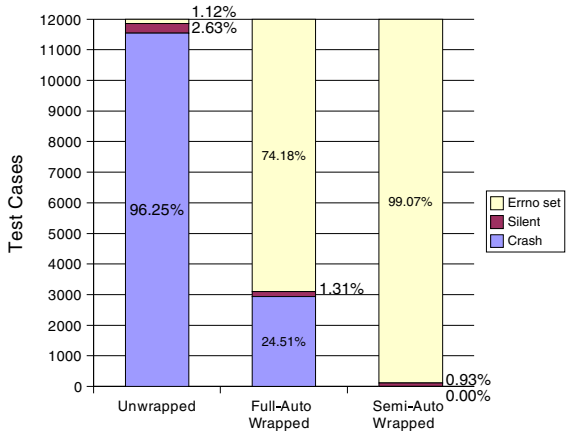


**Figure 6. Test results of 11995 Ballista tests for 86 functions.**

2.0.18 [6]. We downloaded all 11995 test programs for which these functions exhibit robustness violations from the Ballista project web site and rerun these programs under Linux 2.4.4 and *glibc2.2*. The results are shown in Figure 6. Only 9 functions never crash. All other 77 functions crashed for at least one test case.

We used our system to extract type information for all these functions and generated a fault-injector for each of them. Inspecting the robust argument types derived by the fault-injector, we discovered a few interesting things. For example, while function `cfsetispeed` (sets the input baud rate) only needs write access to its argument, function `cfsetospeed` (sets the output baud rate) needs both read and write access to its argument. In addition, functions `fopen` and `freopen` crash when the mode string is invalid but can cope with invalid file names.

We also computed error return code for these functions based on fault-injection experiments. The results are shown in Table 1. We manually examined the results returned by the generated fault-injectors. In particular, we were interested in the 37 functions for which there was no error code found. Our system can automatically detect the error return code if a function sets `errno`. Only one of these 37 functions, `fflush`, is supposed to set `errno`. The two functions that set `errno` inconsistently are `fdopen` and `freopen`: they sometimes set `errno` even though a valid file descriptor is returned.

We created a robustness wrapper from the generated function declarations and evaluated its effectiveness with Ballista. Figure 6 indicates that this automatically generated wrapper reduces the crash failure rates significantly: only 16 functions crashed with the wrapper, versus 77 without the wrapper. The failures that remain undetected usually in-

| Return Code Class | Number | Percentage |
|---|---|---|
| No Return Code | 8 | 9.3% |
| Consistent Error Return Code | 39 | 45.3% |
| Inconsistent Error Return Code | 2 | 2.3% |
| No Error Return Code Found | 37 | 43.0% |

**Table 1. Test results for error return code determination.**

| Applications | tar | gzip | gcc | ps2pdf |
|---|---|---|---|---|
| #wrapped func/sec | 3545 | 43 | 388998 | 378659 |
| time in library | 1.05% | 0.01% | 10.20% | 7.96% |
| checking overhead | 0.16% | 0.0003% | 1.72 % | 1.88% |
| execution overhead | 3.14% | 1.12% | 16.1% | 5.67% |

**Table 2. Execution overhead of four utility programs.**

volve corrupted data structures in accessible memory. Checking the integrity of such data structures requires the wrapper to keep state information, a task our system cannot automate at this stage. One example is the `closedir` function described in the previous section that requires its argument be a directory pointer returned by a previous call to `opendir` function.

In the next step, we manually edited the generated function declarations to add robust argument types and some executable assertions (which we used to track directory structures). With these additional checks we were able to eliminate all crash failures in the Ballista test as shown in Figure 6. This does not necessarily mean that the resulting system will never crash in practice, because Ballista tests do not cover all possible failure scenarios. Nevertheless, it demonstrates the effectiveness of our automated technique in preventing a broad range of robustness violations.

## 7. Performance Overhead

In this section, we evaluate the performance overhead of our robustness wrapper for four utility programs: *tar*, *gzip*, *gcc*, and *ps2pdf*. For each program, we compute the percentage of execution time spent in argument checking. We also compare the overall execution time with and without the wrapper. As described earlier, our wrapper generator can be configured to generate a variety of wrappers for different purposes. In order to better understand the causes of performance overhead, we also generated a measurement wrapper that measures the frequency of function calls for the four utility programs and the percentage of their execution times that is spent in the wrapped **C** library. The results are shown in Table 2.

As can be seen from the table, there is a substantial difference among the behaviors of the four programs. For example, *gcc* spends far more time in the wrapped function calls than *gzip*. Consequently, it incurs a higher execution overhead. Another reason for its high overhead is that *gcc* creates five processes and hence incurs the overhead of loading the wrapper multiple times. Given the extensive error checking that were performed, we believe that this overhead is reasonably small. Further improvements can be achieved using the caching techniques to check the validity of pointer as described in [3].

## 8. Related Work

Fault-injection experiments were previously used in the Ballista system to evaluate the robustness of POSIX operating systems [8, 6]. In their approach, various combinations of valid and invalid input values are generated automatically based on argument types and are fed into **C** functions to see whether exceptional conditions are handled correctly. They found that many implementations of **C** libraries are not robust and may crash or hang due to invalid inputs. Moreover, different implementations of **C** libraries are not completely diverse and may exhibit common failures [6].

Both Ballista and HEALERS use automated fault-injection experiments to discover robustness problems in **C** libraries. However, the two systems are different in significant ways. The Ballista testing methodology requires as input the prototypes of POSIX functions. For a given function, it discovers a list of function calls that exhibit robustness violations. In contrast, the HEALERS system extracts function prototypes automatically by parsing header files and manual pages. It constructs a hierarchy of types and uses adaptive fault-injection experiments to compute the *robust* argument types for a function instead of enumerating all test cases that causes the function to crash. The generated function declarations (with some manual editing) are then used to produce a robustness wrapper that prevents all crash or hang failures in the Ballista tests.

Previously, software wrappers have been used for fault-tolerance [10] and exception handling [11]. Xept is a software instrumentation tool that can be used to handle exceptions from library functions [11]. It provides a language to write exception specifications for certain **C** functions as well as a convenient framework to incorporate such specifications into application code. The advantage of our approach is that the generation of function type information and exception specifications (more importantly, how to avoid exceptions) is highly automated. In addition, the wrapper generator can be configured to generate a variety of wrappers to suite the need of application programmers.

The invalid input handling capability of **C** functions in I/O libraries have been evaluated in [2] using the Ballista system. Their results show that even robust libraries like SFIO [7] may still fail due to invalid file parameters or corrupted data structures. The authors of [2] manually coded safe versions of eight functions. As in our system, each function performs argument checking before calling the original function. However, their system tests memory accessibility using a signal handler: the system touches the memory to see whether it generates memory access fault. If so, the signal handler will catch the exception and the safe function can return an error code. In contrast, our system avoids the usage of signal handlers for memory allocated on the heap and on the stack. An advantage of our approach is that it can detect buffer overflows that occur within the same memory page. Such overflows typically do not generate memory access fault and hence cannot be detected using a signal handler. Nevertheless, they may overwrite other data structures after the buffer and may impose security risks in some systems [4, 1].

## 9. Conclusion and Future Work

Software robustness is essential to critical applications. This paper presents an automated approach to increase the robustness of **C** libraries through adaptive fault-injection experiments. Our system extracts function prototypes in a shared library through header files and manual pages. It then generates a fault-injector based on a carefully crafted type hierarchy to test the *robust* argument types of each global function. Based on this information and some manual editing, the system generates a robustness wrapper that prevents all crash or hang failures in the Ballista tests.

Our approach can improve the robustness and security of libraries that are only available as binaries. It also provides transparent protection to existing programs without modification or recompilation of the source code. In addition, the highly automated nature of the wrapper generation process makes it easy to adapt to new library releases.

In the future, we plan to evaluate the robustness of our system using other types of fault injection techniques (e.g. bit-flips) and its effectiveness with respect to real failure occurrences.

## Acknowledgments

We are grateful to Trevor Jim, Karin Hogstedt, Bob Gruber, and Joao Gabriel Silva for their helpful comments and suggestions. We would also like to thank the anonymous reviewers for comments on an early draft of the paper.

## References

[1] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of USENIX Annual Technical Conference*, June 2000.

[2] John DeVale and Philip Koopman. Performance evaluation of exception handling in I/O libraries. In *Proceedings of the International Conference on Dependable Systems and Networks*, July 2001.

[3] John P. DeVale and Philip Koopman. Robust software – no more excuses. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[4] 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.

[5] Masaharu Goto. CINT C/C++ interpreter, available at http://root.cern.ch/root/Cint.html.

[6] Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, Sep 2000.

[7] David G. Korn and K. Phong Vo. Sfio: Safe/fast string/file IO. In *Proceedings of USENIX Conference*, pages pp 235–256, 1991.

[8] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pages 30–37, June 1998.

[9] Sun Microsystems. Linker and libraries guide, July 2001.

[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] 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.