Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

Tools to make C programs safe: a deeper study*

WANG Ji-min (王继民)†1, PING Ling-di (平玲娣)[1], PAN Xue-zeng (潘雪增)[1]
SHEN Hai-bin (沈海斌)[2], YAN Xiao-lang (严晓浪)[2]
([1]*School of Computer Science, Zhejiang University, Hangzhou 310027, China*)
([2]*Interdisciplinary Research Center on System on the Chip, Zhengjiang University, Hangzhou 310027, China*)
†E-mail: bigjim@zju.edu.cn

Received Oct. 19, 2003; revision accepted Jan. 12, 2004

Abstract: The C programming language is expressive and flexible, but not safe; as its expressive power and flexibility are obtained through unsafe language features, and improper use of these features can lead to program bugs whose causes are hard to identify. Since C is widely used, and it is impractical to rewrite all existing C programs in safe languages, so ways must be found to make C programs safe. This paper deals with the unsafe features of C and presents a survey on existing solutions to make C programs safe. We have studied binary-level instrumentation tools, source checkers, source-level instrumentation tools and safe dialects of C, and present a comparison of different solutions, summarized the strengths and weaknesses of different classes of solutions, and show measures that could possibly improve the accuracy or alleviate the overhead of existing solutions.

Key words: Unsafe feature, C language, Instrumentation tools, Safe dialect
doi:10.1631/jzus.2005.A0063 **Document code:** A **CLC number:** TP314

INTRODUCTION

The C programming language is expressive and flexible, and provides precise and low level control over the program data. However, the expressive power and flexibility of C are obtained through unsafe language features, including unrestricted use of pointers (pointer arithmetic and conversion between pointers and integers), unchecked type casts, explicit memory management, etc. Improper use of these features can result in unexpected program behaviors, the causes of which are often hard to identify because these bugs cannot be reliably reproduced (Burrows *et al*., 2003).

Safe languages such as Java, ML, and Modula-3 have built-in facilities to ensure safety, which make them good candidates to be used to construct system applications. However, they are not suitable solutions for everyone. For one thing, as there exist innumerable lines of C code in current operating systems and all kinds of applications, it is not a sensible idea to abandon the existing C programs and rewrite them from scratch using safe language. For another, the safe languages may not be capable of the task because they do not offer the programmer control over low level data operations (Jim *et al*., 2002). Thus we must find an alternative solution to this problem, keep the high performance of C language, and at the same time, make it safe.

Many solutions have been proposed recently, with different aspects of the safety problems taken into account. This paper aims to provide a survey on the recent development in this area.

UNSAFE FEATURES OF C

Errors in compiled C programs can be classified into two classes: memory access errors and type errors. A memory access error occurs when a program

*Project (No. 2003AA1Z1060) supported by the National Hi-Tech Research and Development Program (863) of China

accesses an invalid memory location; a type error occurs when an operation is performed on operands whose types are incompatible with the operation (Burrows *et al*., 2003). It is believed that the reasons why these errors show up in C programs reach deeper than just poor training and effort: they have their roots in the design of C itself (Jim *et al*., 2002).

Memory access related

(1) Unchecked memory access

Not a single memory access is checked automatically in C. C leaves it to the OS to judge whether it violates memory safety and then whether the executing program should be terminated. Memory accesses are checked by the OS only at the boundary between images of programs and between OS and program images. Memory access errors that do not go beyond the boundary are simply let alone but may damage the program data and cause the program to terminate unexpectedly, leaving little or no useful information to the programmer. These errors may not be reliably reproduced and their causes are hard to isolate and identify.

(2) Unrestricted pointer arithmetic

Unrestricted pointer arithmetic may generate pointers to any location of the memory. If a pointer to a buffer goes out of bounds, and is dereferenced, the program data or even the program code may be damaged. This is the notorious "buffer overflow". Buffer overflows are often caused by misuse of pointer arithmetic. Arrays in C are treated just as pointers, so array access by a bad subscript can also generate such an error. Off-by-one errors are of this kind.

(3) Explicit memory management

The programmer can explicitly allocate a piece of memory when needed, and free it when the need is over. However, it often leads to memory leaks or second-free errors (trying to free a piece of memory more than once) when the free calls does not match the allocation calls.

(4) Null-terminated representation of strings

Strings in C are represented as null-terminated byte arrays. The length of a string is not stored explicitly in the string but is calculated dynamically by finding the terminating zero. This often leads to string manipulation errors (Dor *et al*., 2001). 60% of the UNIX failures reported by the fuzz study (Miller *et al*., 1995) in 1995 were due to string manipulation errors.

Type related

(1) Arbitrary type casts

C allows arbitrary usage of type casts. Casts between integers and pointers, and between various types of pointers are commonplace in C. There are also cases where the type of the operand does not match the operation, yet the compiler either knows nothing about this or is made dumb by an explicit type cast. Type errors often cause dynamic errors.

(2) Variable argument functions

In such a function, the compiler does not know what type each argument should be, so takes whatever for granted, even does not know the precise number of arguments each function call should have. Statements like

```
int i, j;
scanf("%d %d", i, j);
```

and

```
printf("%s");
```

will get no error messages during compiling. Misuse of these functions can cause the notorious "format string vulnerabilities" (Bouchareine, 2000; Scut, 2001).

(3) Union

Different members of a union mean different representations of the same piece of memory. Writing one member of the union and reading another sometimes can cause complicated errors:

```
typedef union {
    char c[4];
    float f;
}_u;
_u u;
u.c[0] = 1;
printf("%f",u.f);
/* uninitialized memory accessed! */
```

(4) Function pointers

C provides function pointers but provides no guarantee that the function pointer exactly matches the function in arguments and returned types. If wrong arguments are passed to a function through the mismatched function pointer, or wrong type of data are returned, errors may show up.

EXISTING SOLUTIONS

Binary-level instrumentation tools

Binary-level instrumentation tools can be used to

check or inspect almost all runnable programs in the system. No recompilation is needed–even the source is not needed. There are no compatibility problems between the programs and libraries they use–the analysis tool treats them alike, and reports all detected errors, regardless of what namespace they come from.

Purify (Hasting and Joyce, 1992) maintains a two-bit state code for each byte of user memory. This two-bit state code records the current state of the corresponding byte. At the same time, Purify modifies the program by inserting a function call instruction into the program's object code before every load and store, and the called function will perform the memory safety check. To catch buffer overflow violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by malloc and marks it as unallocated (unwritable and unreadable). Valgrind (Seward, 2003) creates a synthetic CPU at program start up, translates the program code first into self-defined intermediate code block by block, instruments it and optimizes it, then translates it back into x86 code, and executes it on this CPU. The intermediate code is instrumented to do necessary checking, so it can detect memory access errors. Valgrind maintains two types of state code to record the validity of memory and uses these bits to detect memory access errors. To detect memory management related errors and memory leaks, Valgrind provides its own version of memory management functions such as malloc and free. Hobbes (Burrows *et al*., 2003) is implemented as an interpreter. It fetches instructions from the target instruction stream and performs the corresponding operation. Hobbes maintains a shadow memory for the main memory, in which the memory status (initialized or not) and type-related information are recorded. Each instruction fetched will be type checked by a specific instruction analysis routine based on the information kept in the shadow memory.

Static source check tools

A static check tool statically checks C programs to find safety vulnerabilities and coding mistakes. It depends on no more than the program source and behaves just like a C compiler, but can find far more errors.

Flawfinder (David, A., 2003) and ITS4 (Viega *et al*., 2000) used a built-in database of C/C++ functions with well-known problems to help the programmer improve his program. BOON (Wagner *et al*., 2000; David, W., 2003) treats C strings as abstract data type and models buffers as pairs of integer ranges. It first parses the source program, associates necessary range variables to each program variable and generates an integer range constraint for each statement of the input program, then constructs a directed graph from the constraints and uses a solver to traverse the graph, checking the safety property of each string, and reporting possible buffer overflows. SpLint (Larochelle and Evans, 2001; Evans, 1996; 2003) is used to detect anomalies in C programs. By simply analyzing the unmodified source code, Splint can detect many logical errors and it can be even more powerful by exploiting annotations added to libraries and programs. PREfix (Bush *et al*., 2000) analyzes a program by simulating the execution of individual functions on an underlying virtual machine. It first parses the source code into abstract syntax trees, then for each achievable path through the function, it traverses the function's abstract syntax tree and evaluates the relevant statements and expressions in the tree. The behavior of a function is embodied by a model and when the simulator encounters a function call, it emulates the called function by using the function's model.

Source level instrumentation tools

Source level instrumentation tools insert necessary checking statements into the program source or add extensions to C syntax to achieve program safety. They rely on the program source to function properly, and the safety checking is often performed at run-time.

Safe-C (Austin *et al*., 1994) changes the representation of a pointer to a 5-tuple: value, base, size, storageClass and capability. It can detect temporal memory access errors as well as spatial memory access errors. Because pointers are not the normal size, they need "encapsulation" when doing system calls and calling non-instrumented functions. Jones and Kelly presented a backwards-compatible bounds checking method and implemented it in GCC (Jones and Kelly, 1997). In their implementation of GCC, all known valid storage objects are maintained in a table, and one can use the table to map a pointer to a descriptor of the object into which it points, which

contains the base, extent and additional information to improve error reporting. Because a pointer in new representation is of the same size as an ordinary one, checked code can inter-operate without restriction with unchecked one. Loginov *et al*. instrumentation tool (Loginov *et al*., 2001) also employs unchanged representation of pointers. It maintains a mirror of user memory at run time, each byte of user memory maps to a four-bit nibble in the mirror, indicating the dynamic type of the memory. This tool can catch type errors as well as memory access errors. Pointers in CCured (Necula *et al*., 2002; 2003; Condit *et al*., 2003) are divided into three categories: safe pointers, sequence pointers and dynamic (or wild) pointers. Each has its own capability (e.g. can it be subject to pointer arithmetic? can it be cast to integer?), and this capability can be used in static or run-time analysis. Tagged union is also provided in CCured to prevent bad union access.

Safe dialects of C

Some tools change the syntax of C and try to make it a safe language. The new language is safer but is not compatible with the original C and existing C programs must be modified to compile in the new environment. This is not always a good idea since some programs contain millions of lines of C code and it may take a long time to transplant them to the new C dialect.

Cyclone (Jim *et al*., 2002; Grossman *et al*., 2002) improves security of C by imposing restrictions on the original C and adding extensions to it. There are three kinds of pointers in Cyclone: ordinary pointers (*), never-NULL pointers (@) and fat pointers (?). Pointer arithmetic is allowed only on fat pointers. Pointer safety is ensured by inserting null checks before dereferencing never-NULL pointers and ordinary pointers and by inserting bounds checks before dereferencing fat pointers. Cyclone extends the union type by adding a tag to each case of the union. Type-varying arguments in Cyclone are treated as tagged-unions.

A DEEPER STUDY

We applied all available tools mentioned above to a set of small programs to evaluate the performance of these tools; the result is shown in Table 1 through Table 3. All tested tools are the latest version available on the Internet: PurifyPlus 2003a.06.00, CCured 1.2.1, Valgrind 2.0.0, Cyclone 0.6, SpLint 3.1.1, boundschecking-gcc (Jones and Kelly tool) 3.3.1-1.01. All testing programs are compiled with gcc 3.3.1 and optimized with its -O2 option. All data are collected on a 1.7 GHz Celeron4 with 512 MB of memory, running Mandrake Linux 9.2 (kernel 2.4.22), except Purify, since PurifyPlus 2003a.06.00 for Linux

Table 1 Performance measurements for different sorting algorithms and the maze problem

Program	Base time		Purify		Valgrind		CCured		Jones		Cyclone		Loginov	
	Linux	SunOS	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
Insertion1	1.99	12.44	313.45	25.20	55.77	28.03	26.89	13.51	117.90	59.25	8.15	4.10	598.92	300.96
Insertion2	2.05	8.31	165.60	19.93	58.36	28.47	26.97	13.16	77.98	38.04	6.02	2.94	545.83	266.26
Quick	0.01	0.02	0.29	14.50	0.14	14.00	0.04	4.00	0.12	12.00	0.02	2.00	0.93	93.00
Shell	2.80	17.34	388.02	22.38	62.54	22.34	22.11	7.90	168.02	60.01	8.36	2.99	648.14	231.48
Selection	3.48	16.78	263.39	15.70	75.94	21.82	27.42	7.88	79.13	22.74	6.35	1.82	412.96	118.67
Bubble	9.83	33.45	630.44	18.85	163.27	16.61	92.55	9.42	321.29	32.68	28.74	2.92	1808.24	183.95
Merge1	0.01	0.03	0.59	19.67	0.29	29.00	0.10	10.00	0.28	28.00	0.03	3.00	1.43	143.00
Merge2	0.98	0.15	9.11	60.73	91.78	93.65	113.43	115.74	1.05	1.07	1.02	1.04	40.75	41.58
Heap	0.02	0.01	0.58	58.00	0.36	18.00	0.09	4.50	0.26	13.00	0.03	1.50	1.91	95.50
Maze	3.47	16.02	64.66	4.04	60.58	17.46	13.87	4.00	51.86	14.95	N/A	N/A	104.50	30.12
Slowdown	–	–	4.04–60.73		14.00–93.65		4.00–115.74		1.07–60.01		1.04–4.10		30.12–300.96	
Median	–	–	25.90		28.94		19.01		28.17		2.48		150.45	

All times are in seconds. The ratio columns indicate performance slowdowns relative to the base time. All tools run on Linux except purify, so we list two base times in the table

has problems while running on the Linux platform. Purify runs on a dual-processor (Ultra Sparc-II 450 MHz) Sun Ultra 60 workstation with 2 GB of memory, running SunOS 5.8.

Execution time First we applied all available source instrumentation tools and binary-level instrumentation tools to implement different sorting algorithms and the maze problem (Cyclone is also included since it is easy to translate these C programs into cyclone language), and then recorded the execution time of each tested program. The result is shown in Table 1. Among the sorting algorithms, "Insertion1" is direct insertion sorting algorithm, "Insertion2" is binary insertion sorting algorithm, "Merge1" is the iterative version of mergence sorting algorithm, "Merge2" is the recursive version of mergence sorting algorithm, and others are self-explaining.

It does matter to interpret the code or execute it directly. Purify incurs a slowdown of about 26 times over the base time, Jones and Kelly tool incurs a slowdown of about 28, and Valgrind incurs a slowdown of about 29. There is no other apparent difference among them. However, Hobbes, which interprets programs, incurs a slowdown of 72 (MemCheck) or 141 (TypeCheck) (Burrows *et al*., 2003). To some extent, the way how the code is executed decides the performance of the tool. The overall performance of CCured is not much better than that of these three tools, however, if we leave "Merge2" aside, things will change greatly: the median slowdown incurred by CCured will be 8.26 times over the base time. Valgrind also has this problem. This may imply that CCured and Valgrind can be improved by changing the way they instrument recursive functions. Among all tested tools, Loginov *et al*. tool performed worst as it did type checking as well as memory checking, while other tools did memory checking only. Cyclone had the best performance among all the tools, which may attributed to its restrictions on C and built-in safe features.

Memory overhead To study the memory overhead of these tools, we recorded the maximum size of memory occupied by the tested program each time we applied a tool to the program and run it (the result is shown in Table 2). Valgrind incurred the greatest memory overhead because it maintains a one-bit state code for each bit of memory. Purify maintains a two-bit state code for each byte of memory, and Loginov *et al*. tool maintains four bits for each byte of memory, so their memory overhead was also high. The fat representation of pointers of CCured incurs an increase of about 2 times over the base size.

Error detection We wrote a small test suite to evaluate the error detection ability of different tools. The test suite was composed of a number of C program files, each containing one or more errors caused by improper use of unsafe features of C. The result is listed in Table 3 showing that binary-level instrumentation tools and source instrumentation tools are more sensitive to buffer overflows, and generally do a better job in detecting buffer overflows and memory management errors than source checkers. However, none of these tools detected the error of reading uninitialized locals or reading uninitialized non-buffer objects on heap. SpLint did a better job this time. As for function pointer related errors and vararg related errors, almost all tools saw nothing wrong about them and could not offer any help to the programmer.

CONCLUSION

It is necessary to make the C programming language safe since many important software systems are written in C, and there are still more to come. The unsafe features of C should be resolved with the help of safety tools which should be able to detect memory access errors as well as type errors accurately and quickly; and the overhead introduced to the original program should be as low as possible.

Source check tools work quickly and introduce no overhead to the checked program. By checking the source they can detect many logical programming errors such as unreachable code and infinite loops. However, they do not have access to the dynamic program information so they cannot detect dynamic memory access errors, and often give too many misleading error reports. Binary-level instrumentation tools do not depend on the program source to work, which makes them applicable to a variety of programs. However, they cannot optimize memory checks and type checks since they do not have access to extra information provided by the source, so the overhead introduced by these tools is generally high. What is more, working at binary level limits them to a single

Table 2 Memory overhead measurements for different sorting algorithms and the maze problem

Program	Base size		Purify		Valgrind		CCured		Jones		Loginov	
	Linux	SunOS	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio
Insertion1	852	1232	6056	4.92	10772	12.64	1212	1.42	944	1.11	3556	4.17
Insertion2	852	1232	6056	4.92	10776	12.65	1216	1.43	952	1.12	3560	4.18
Quick	860	1232	6072	4.93	10776	12.53	1216	1.41	960	1.12	3564	4.14
Shell	860	1232	6072	4.93	10776	12.53	1216	1.41	948	1.10	3556	4.13
Selection	852	1232	6072	4.93	10776	12.65	1216	1.43	948	1.11	3560	4.18
Bubble	860	1232	6056	4.92	10772	12.53	1212	1.41	948	1.10	3556	4.13
Merge1	1052	1416	6256	4.42	10964	10.42	1216	1.16	1148	1.09	3752	3.57
Merge2	1044	4544	9400	2.07	20344	19.49	8488	8.13	1144	1.10	7372	7.06
Heap	860	1232	6072	4.93	10776	12.53	1216	1.41	948	1.10	3560	4.14
Maze	16752	33456	45120	1.35	79016	4.72	33328	1.99	33572	2.00	35592	2.12
Augmentation	–	–	1.35–4.93		4.72–19.49		1.16–8.13		1.09–2.00		2.12–7.06	
Median	–	–	4.23		12.27		2.12		1.20		4.18	

All sizes are in kB. The ratio columns indicate memory occupation augmentation relative to the base size. All tools run on Linux except Purify, so two base sizes are listed in the table

Table 3 The result of applying different tools to a test suite

Error description	Purify	Valgrind	CCured	SpLint	Jones	Loginov
Reading uninitialized locals	–	–	–	+	–	+
Reading uninitialized integer on heap	–	–	–	+	–	+
Reading uninitialized string on heap	+	+	–	+	–	+
Writing overflowed buffer on stack	–	–	+	–	+	+
Writing overflowed buffer on heap	+	+	+	*	+	+
Writing to unallocated memory	+	–	–	+	+	+
Returning stack object	+	+	+	+	+	+
Overwriting ending zero of string	–	–	–	–	–	–
Function pointer with wrong number of arguments	–	–	–	–	–	+
Function pointer with wrong returning type	–	–	–	–	–	+
Vararg with wrong type of arguments	–	–	–	+	–	+
Vararg with wrong number of arguments	–	+	+	+	–	–
Bad union access/part of an object is uninitialized	–	–	–	–	–	+
Bad union access/a complete uninitialized object	+	–	–	+	–	–
Memory leak	+	+	–	*	–	–
Second free	+	+	+	*	+	–
Casting integer literal to pointers	–	–	–	+	–	+
Casting floating pointers to integer pointers	–	–	+	+	–	+

"Error description" describes a kind of error contained in one or more programs; "+" indicates that the corresponding tool detects this kind of error correctly; "–" indicates that the corresponding tool cannot detect this kind of error correctly; "*" indicates that the corresponding tool detects this kind of error correctly in some programs, while failing to detect it in others

OS platform since binaries of different OS platforms are generally different. Source-level instrumentation tools have full access to the program information, which makes it possible for them to detect all memory access errors and type errors, and the overhead can be reasonable if full optimization is made and only necessary checks are performed. Another merit of source level instrumentation tools is that they can be used on different OS platforms. Safe dialects of C also have the merits that the source-level instrumentation tools have. By imposing restrictions on the original C and by introducing new safely features to it, they can even achieve better performance. However, they are not compatible with the original C, so they cannot be used to "purify" existing C programs, or at least efforts must be made to transform existing programs to the new environment.

Future safety tools will still focus on accuracy and overhead. To improve accuracy of memory access checking, we must perform the analysis at bit level. Currently almost all checking tools work at byte level, but not at bit level (Valgrind sets a one-bit state code for each bit of user memory and registers, but the definedness of memory and registers is not really maintained at bit level). The reason why they do not do that is unique: if status and type information are maintained at bit level, the memory overhead of the checking tool will be intolerably high. However, bit fields do exist in the C Language, and their status and type information cannot be represented completely and precisely at byte level. The solution is to divide and conquer: represent extra information of integers, pointers, floating numbers and non-bit-fields members of structures at byte level, and represent bit-fields members of structures at bit level. So where to store these extra information? Use a shadow memory to store the byte-level information, and set up direct mapping between the shadow memory and the main memory, so the extra information can be quickly addressed. Allocate another piece of memory–which we call reference memory–to store the bit-level information. Extra information on bit fields may be indexed by its address to accelerate the accessing speed. The state code of bit fields in the shadow memory is set to a special value, indicating that the needed information is stored in reference memory. We need two addressing to access the extra information of bit fields, however, these accesses are minor among all memory accesses, and the overhead is tolerable.

To improve the accuracy of type checking, a model must be set up to address clearly what is a type error and what is not. However, this is not easy because the type system of C is so subtle. For example:

```
1  typedef union {
2          char c[4];
3          float f;
4  }_u;
5  _u u;
6  u.f = 1.234;
7  printf("%c", u.c[3]);
```

Should the access of u.c[3] in line 7 be reported as a type error? Neither "Yes" nor "No" is the perfect answer. So false alarms and wrong error reports are inevitable. What we can do is to detect as many errors as we can, and minimize the number of false alarms and wrong error reports. Studies of C type systems (Smith and Volpano, 1998; Siff *et al*., 1999; Chandra and Reps, 1999) can offer us great help.

To minimize the overhead introduced to the original program, an optimization must be made to minimize the checks added to the program, which may require a deeper study on the C program behavior. There are also studies on this issue (Jagannathan and Wright, 1995; Bodik *et al*., 2000; Arnold and Ryder, 2001).

References

Arnold, M., Ryder, B.G., 2001. A Framework for Reducing the Cost of Instrumented Code. Proceedings of the Conference on Programming Language Design and Implementation(PLDI), Salt Lake City, p.168-179.

Austin, T.M., Breach, S.E., Sohi, G.S., 1994. Efficient Detection of All Pointer and Array Access Errors. Proceedings of the Conference on Programming Language Design and Implementation (PLDI), p.290-301.

Bodik, R., Gupta, R., Sarkar, V., 2000. ABCD: Eliminating Array Bounds Checks on Demand. SIGPLAN Conference on Programming Language Design and Implementation(PLDI), p.321-333.

Bouchareine, P., 2000. Format String Vulnerability. Bugtraq. http://www.hert.org/papers/format.html

Burrows, M., Freund, S.N., Wiener, J.L., 2003. Run-time Type Checking for Binary Programs. International Conference on Compiler Construction.

Bush, W.R., Pincus, J.D., Sielaff, D.J., 2000. A static analyzer for finding dynamic programming errors. *Software, Practice, and Experience*, **30**(7):775-802.

Chandra, S., Reps, T., 1999. Physical Type Checking for C.

Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, volume 24.5 of Software Engineering Notes (SEN), p.66-75.

Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W., 2003. CCured in the Real World. Proceedings of the Conference on Programming Language Design and Implementation (PLDI).

David, A., 2003. Flawfinder Documentation. http://www.dwheeler.com/flawfinder/.

David, W., 2003. Boon Home Page. http://www.cs.berkeley.edu/~daw/boon/.

Dor, N., Rodeh, M., Sagiv, M., 2001. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. 8th International Symposium on Static Analysis (SAS), p.194-212.

Evans, D., 1996. Static Detection of Dynamic Memory Errors. SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

Evans, D., 2003. SpLint Documentation. http://www.splint.org/.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y.L., Cheney, J., 2002. Region-based Memory Management in Cyclone. ACM Conference on Programming Language Design and Implementation, Berlin, Germany, p.282-293.

Hasting, R., Joyce, B., 1992. Purify: Fast Detection of Memory Leaks and Access Errors. Proceedings of the Winter USENIX Conference.

Jagannathan, S., Wright, A., 1995. Effective Flow Analysis for Avoiding Run-time Checks. Proceedings of the Second International Static Analysis Symposium, **983**:207-224.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.L., 2002. Cyclone: A Safe Dialect of C. USENIX Annual Technical Conference, Monterey, CA, p.275-288.

Jones, R.W.M., Kelly, P.H.J., 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. Proceedings of Third International Workshop on Automated Debugging, p.13-26.

Larochelle, D., Evans, D., 2001. Statically Detecting likely Buffer Overflow Vulnerabilities. 10th USENIX Security Symposium. Washington D.C.

Loginov, A., Yong, S.H., Horwitz, S., Reps, T., 2001. Debugging via Run-time Type Checking. Proceedings of the Conference on Fundamental Approaches to Software Engineering, p.217-232.

Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J., 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report.

Necula, G.C., McPeak, S., Weimer, W., 2002. CCured: Type-safe Retrofitting of Legacy Code. Proceedings of the Symposium on Principles of Programming Languages, p.128-139.

Necula, G., McPeak, S., Weimer, W., Harren, M., Condit, J., 2003. CCured Documentation. http://manju.cs.berkely.edu/ccured/.

Scut, 2001. Exploiting Format String Vulnerabilities. http://teso.scene.at/articles/formatstring/.

Seward, J., 2003. Valgrind, An Open-source Memory Debugger for x86-GNU/Linux. Technical Report, http://valgrind.kde.org/.

Siff, M., Chandra, S., Ball, T., Kunchithapadam, K., Reps, T., 1999. Coping with type casts in C. *Lecture Notes in Computer Science*, **1687**:180-198.

Smith, G., Volpano, D., 1998. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, **32**(13):49-72.

Viega, J., Bloch, J.T., Kohno, Y., McGraw, G., 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. Proceedings of the Annual Computer Security Applications Conference.

Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A., 2000. A First Step toward Automated Detection of Buffer Overrun Vulnerabilities. Network Distributed Systems security Symposium, p.1-15.