

An Analysis of Static Analyzer Tools

Paul DiPalma and Elise Hewett

PROJECT SUMMARY

Traditional methods of debugging programs, such as manual inspection and testing, often result in code that is plagued by memory leaks, null pointers, dangerous aliasing, and the use of constructs that may introduce security weaknesses. Software analysis and debugging tools have been developed which hope to address this problem; however, we do not know how effective these tools are in practice with respect to each other.

The basis of our approach is to apply several analysis and debugging tools such as Valgrind, Splint, ITS4, Electric Fence, and MemWatch to Reliability Block Diagram Solver programs. This contribution involves running the analysis tools on the programs and then comparing the number and types of bugs and security weaknesses that the different tools find in each program. The analysis tools' ability to statically find bugs (e.g. buffer overflow, null pointer referencing or dereferencing, failing to release an acquired lock, and simple typos) and the use of functions with potential security risks (e.g. C's `strcpy`, `gets`, `select`, `stat`, and open functions) will be quantitatively compared by examining how many bugs each tool finds and how many of these bugs (if any) are actually false positives. In addition, qualitative analysis will be performed on several aspects of each tool. We will compare the difficulty in learning to use the tool, the ease of use of the user interface, the helpfulness of the tools' error messages, and whether the tool can be applied to more than just one language.

Since an RBD solver relies on complex data structures and, depending on the input diagram, is forced to manipulate the data in complex ways, it will make a formidable test case for the tools at hand. A helpful tool should find bugs that would inhibit the input data from staying intact as the RBD solver performs its analysis, i.e. checking for memory leaks, buffer overflows, and other terminal bugs.

Our work seeks to integrate the use of automated static software analysis tools into standard software development practices, and thereby reduce the amount of time programmers dedicate to debugging.

Intellectual Merit

Software designers often do not understand the capabilities and limitations of debugging and analysis tools. This work will address several problems that impede the use of such tools.

The first problem that this research aims to address is that programmers do not know of the existence of automated debugging tools or which tools will be the most helpful in solving the problems at hand. Developers continue to rely on traditional methods of debugging even though these tools are readily available.

A final, inherent problem is that programmers simply do not see the cost/time benefit in using automated debugging and analysis tools. As deadlines loom closer and programs are still unfinished, time pressures lead many programmers to perform inadequate, ad hoc quality assurance. Software engineers would rather deploy their project on-time while it still contains bugs and then distribute patches and updates later. This research will provide a quick and easy way for developers to learn about several debugging tools that will save time and money in a software project.

Broader Impact

This proposal seeks to address the lack of information on the effectiveness of software analysis tools. We aim to make the information we gather about the benefits of using these analysis tools (in addition to a listing of the existing tools) widely available to software developers. The overall effect of this work is that any commercial developer can learn about and then hopefully use an appropriate tool in order to save both time and money. Programmers in purely educational settings will also benefit by saving time and increasing the quality of their programs.

An Analysis of Static Analyzer Tools

Paul DiPalma and Elise Hewett

PROJECT DESCRIPTION

Software static analyzer tools are programs that are applied to software systems to look for errors or security vulnerabilities in the code. The use of such software analyzer tools is not frequently encouraged in educational settings when programmers are beginning to develop their programming habits. System designers in industrial settings may be searching for a static analyzer tool that is a better fit for the types of code they develop.

Simpler forms of testing and debugging continue to be the most widely used methods of validating programs. Problems with testing lie in the fact that few large programs can truly be exhaustively tested. An unanticipated path of execution not included in the testing suite could cause the system to fail. Manually checking programs for errors fails due to the tedious and time-consuming nature of the process. Additionally, errors often go undetected and may never be found because the programmer is unaware that a problem exists. Particularly in the case of an extremely large system, brute force manual debugging is simply impractical, if not impossible.

In this research, we will develop a list of tools that effectively find bugs and security problems that exist in C/C++ programs, including those which may never be found through manual debugging techniques. The biggest contribution of this study is to educate programmers of the cost/time benefit of using the tools in the debugging phase of software development.

1 Scientific Objectives

While there are many software analysis tools that are readily available, many programmers never use them. We are faced with overcoming programmers' bad habits of relying on ineffective manual debugging and attempting to use test cases to find errors in their code.

A programmer is generally not educated about the security risks that utilizing certain functions can introduce or just may not realize that he is misusing a function [9]. Errors of this nature will not be found through manual debugging and if the test suite fails to test a particular path of execution, the error may not be discovered at all.

The following example of code illustrates this concept of buffer overflow, the most common type of security bug [12].

```
void causeOverflow(char *a) {
    char myBuffer[4];
    strcpy(myBuffer,a);
}

int main() {
    char *c = "Buggy";

    causeOverflow(c);
}
```

[13]

The basic concept of the buffer overflow security bug is that the programmer allocates a certain amount of characters for a user's input and then doesn't check to see if the user inputs more characters than will fit in the allocated space. A skilled attacker can literally hijack the processor's execution path by overwriting the

return address with the address of their program. The attacker’s program can then start an interactive session on the computer with root privileges.

In large systems programming, assumptions the programmer thinks will be true, both before and after a function is called, are often not met. In the following example of a failure to release a lock, the programmer simply does not call the function that releases the lock to allow other threads access to the critical section. Deadlock ensues.

```
void callToRelease(Lock *thelock) {
    /* critical section */
    Lock->Release();
}

void causeDeadlock(Lock *thelock) {
    Lock->Acquire();
    /* critical section */
}

int main() {
    Lock *thelock;
    causeDeadlock(thelock);
}
```

Although it is possible to manually find errors such as those illustrated in the two previous examples, attempting the same task in very large, complex software systems is non-trivial, very time consuming, and likely to miss many errors that may be found by an automated tool. After a tool has been used to analyze a program, the errors must be examined to determine which are true errors and which are actually just false-positives. True errors can then be corrected. Although using a tool to analyze code does not guarantee correctness, the more common errors will hopefully be caught.

The primary challenge in this research will be developing a fair method of ranking the tools and determining what each tools strengths and weaknesses are. Before the tools can be evaluated for their effectiveness at finding errors, we must determine a quantitative approach to “scoring” a tool’s efforts at debugging. Using a qualitative approach in ranking will also be difficult because of the controversy that can arise in deciding which features of a tool are most critical. We will consider many types of situations in which using one tool could be preferable to another.

Additionally, because we want to help software designers make the best choice in selecting an analysis tool, we must determine a cut-off point for when the time costs outweigh the benefits for using the tool. Deciding at what point a programmer might no longer care about errors (e.g. those in a non-safety critical system that will rarely occur) will be an important factor in this cut-off point.

2 Impact

In compiling a listing of effective analyzer tools, we hope to indirectly help programmers improve several aspects of software development. A package of easy to use tools will ease the testing and debugging process for any software engineer. As a result, memory leaks and security holes, as well as other bugs, will be found faster, thus increasing the speed and quality of development for a particular project.

We hope that this research and testing will bring software developers up to speed with respect to the available debugging tools that exist. Since research time for a programmer is relatively small, our research and experimentation will prove to be invaluable and will naturally stimulate the use of helpful debugging tools among programmers.

Because time is short for the student, who juggles several classes and projects at a time, and the programmer, who is barred by a constant workload and strict deadlines, a set package of tools that have been evaluated and approved will be beneficial. Just the fact that they are readily available will appeal to the masses, as people are willing to trade their trust in tools for time.

3 Research Plan

We will read manuals to learn to use each of the five tools and then apply each tool to all of the RBD solver program. The number and types of bugs that each tool finds for each program will be recorded. The bugs in each program will be analyzed to determine if it is actually a bug or just a false-positive.

Quantitative and qualitative comparisons of the tools will be made in order to devise a comprehensive list of analyzer tools. First, the quantitative differences will be noted by the numbers and types of each kind of error that the tools find in the programs. The expected difference in the higher quality of a formally developed relational block diagram solver versus the lower quality of the students' informally developed RBD solvers would be useful in the quantitative comparisons. Second, we should note the qualitative differences in the tools. The clarity of error messages, simplicity to learn, and ease of use of the tools will also be examined. Additionally, we will look at whether the tool actually does discover the types of bugs that it claims to find.

3.1 Formally Developed RBD vs. Informally Developed RBD

We expect the RBD solver with a formal structure to be less error-prone than that of the rushed, inexperienced students. However, we should take into account what these differences are and how they could possibly affect the programs' performance and reliability.

We will present the structure of the formal method for the Reliability Block Diagram, which was implemented by two engineers whose program we will be using in our test plan. This version of the RBD solver will act as a "control" program to compare the relative quality and structure of the informally developed student versions of the RBD solver. The definitions and concepts of formal methods will be presented briefly in the background section of this proposal.

3.2 Tool Performance on RBD solvers and Relative Comparisons

Since we expect the formally developed RBD solver to be more carefully designed than an informally designed RBD solver class project, we expect an effective tool to reflect this difference. It may result that a particular tool will find no errors in the formal structure but finds numerous errors in the informal structure. In a relative comparison, another tool may find errors in the programs that were not found by one of the other tools. This tool will be rated higher as a result.

3.3 Evaluation of Tools With Respect to Ease of Use

One of the most important factors in choosing an appropriate debugging tool for a software engineer is the ease of use of the particular tool used. In our evaluation of the tools we will weigh the overall quality with a large emphasis on the area of "ease of use". Aside from the performance of a debugging tool and its ability to find as many true bugs as possible, the feedback that it gives the user is extremely important in being an effective tool for a programmer.

Besides just emitting where the bugs were found, the tool should provide an adequate message system in which the user can act on the problem at hand, as quickly and easily as possible. Without this the programmer is left to ponder where the problem is coming from and if it is even worth fixing.

3.4 Test Plan

In our evaluation, we will test five tools that are found on the open market. The tools are seen as debugging software and promise to find various errors in any given software project. To validate and rate the effectiveness of a given tool, we will use them on a reliability block diagram solver (RBD).

In order to further decipher the effectiveness of the tool, we will use an informal and formal version of an RBD solver. The informal version was engineered by students for a Stochastic Modeling class at the College of William and Mary, in which time was a major factor and the only goal of the project was to have a working version. There are five separate student projects that will be used, where each project contains between five hundred and one thousand lines of code. The later version is also an RBD solver (about 3000 lines of code), but engineered by two people, David Coppit, Ph.D. and Robert Painter. In the formal version more time allowed for a formal specification to be developed and eventually led to a solid implementation of the RBD solver.

Since these various tools are language specific, we will be focusing on testing RBD solvers that are written in the C or C++ language. Even though these solver programs cover two languages the effectiveness of the tools should provide comparable results.

3.5 The Rating Scheme

In ranking the tools, we should set a relative gauge or score in order to properly compare the effectiveness of each one. Our scale will be in the range of zero to one hundred and will be based on three criteria. Thirty points will be allocated to scale the ‘ease of use’ criteria, the measure of a tool’s relative learning curve as well as ease of acquisition, installation, and/or setup. Thirty more points will be allocated for feedback message ratings, a measure of the verbosity of the messages, and how they help the user to track down the problem. In this case, we will act as the user in order to devise a rating. Finally, forty points, the highest weight, will measure the amount and quality of bugs found by each tool.

4 Background

4.1 Debugging techniques

The most primitive, yet sometimes effective, way that programmers debug their software is through simple “print” statements. The programmer can insert output statements by hand at trigger points, places where he/she believes that there may be an error. Although extremely time consuming this may be the only tool available in the mind of the programmer.

A step up from manual output statement analysis is a simple debugger such as “gdb”, seen in Unix for debugging C programs. In its basic functionality, gdb allows the user to see certain elements of his/her program while it is running. This creates a step by step process that is slowed down to an analytical pace so that a programmer can view any possible problems. This method replaces the tedious placement of output statements, described above, but still requires the user to do most of the analysis in where a problem or bug might originate.

In a truly perfect world, a set of tools would exist that run a given program, debug it, find errors, and either correct the errors or just tell the user where the errors are and how to fix them. This may seem far fetched, but in searching for a set of debugging tools we expect to find tools that will perform better than the basic debugging techniques described above and approach the expectations of our perfect world.



Figure 1: An example of a series reliability block system

4.2 Secure Programming in C

A harmful bug, seen in the C programming language, is a buffer flow error. This can happen if a programmer is not careful to check the inputs and outputs, as well as the memory allocation routines of his/her program. A buffer overflow occurs when programs try to store more data in a variable than it has been allocated space for. A simple example is the use of the C function “gets(var)” instead of “fgets(var,10,stdin)”. The first use of the get function allows for a malicious user to input a lengthy string, longer than 9 characters, that could possibly cause such harm as build on the stack the instructions needed to start a shell [8].

We will analyze several debugging tools that look for the problem described above, as well as many other possible memory and security problems. These bugs must be found and fixed at all costs, but this cost can be reduced with the use of these tools.

4.3 Basic Concepts of RBDs and Problems

A Reliability Block Diagram (RBD) is a visual representation of a system that contains redundancy in its elements. Redundancy is used for mission critical functions where a single-point failure is not acceptable and reliability needs to be improved [6]. The basic elements of an RBD are parts of the system in parallel, series, series-parallel, and even a special case where k-out-of-N elements must work.

Simple RBDs are generally represented in an directed, acyclic graph and their reliabilites may be calculated using analytical solutions. Results may include reliability, availability, failure rate, and mean time before failure. The goal is to improve the field reliability of the system. An RBD solver will simulate the graphical representation of the diagram and calculate the reliability, R, of the given system. The solver acts on a set of rules and formulas that are inherent in calculating reliability.

The rules for calculating the reliability are as follows:

1. Series Components If parts of a system are presented in series, then the reliability of this set of parts is simply the multiplication of the reliability of each component in the series. In Figure 1, if $R_A = \frac{1}{3}$, $R_B = \frac{1}{4}$, and $R_C = \frac{7}{8}$, then the reliability of the system is $R = \frac{7}{96}$
2. Parallel Components If the parts are in a parallel configuration, the reliability of this parallel grouping would be equal to one minus the multiplication of one minus the reliability of each component, so, $R = 1 - \prod_{i=0}^n (1 - R_i)$. In Figure 2, if $R_A = 1$, $R_B = \frac{1}{4}$, $R_C = 1$ and $R_D = \frac{1}{5} = \frac{4}{10}$ then the reliability of the system is $R = \frac{2}{5}$.

These two rules form the basis for calculating the reliability of systems where series and parallel elements are intertwined. One further extension of an RBD is taking into account a *k-out-of-n* reliability factor, which puts a strict limitation on the number of elements in parallel that must work at the same time.

Since the rules for calculating the reliability for a RBD are based on probabilities and in mathematics, we can see how formal methods for creating a solver can be instituted. We will take a brief look at formal methods in the next section.

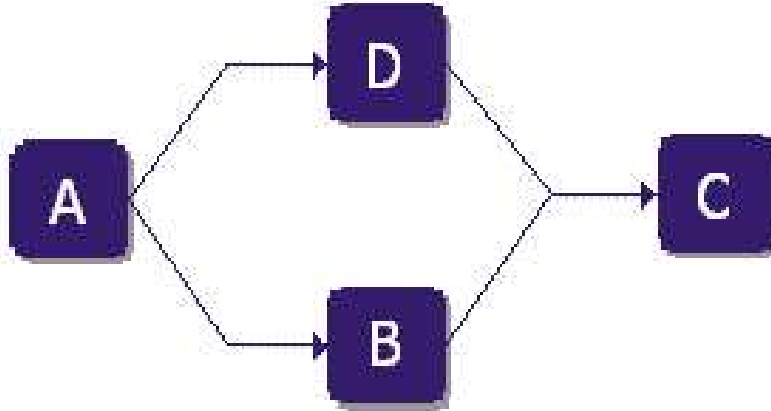


Figure 2: An example of a parallel reliability block system

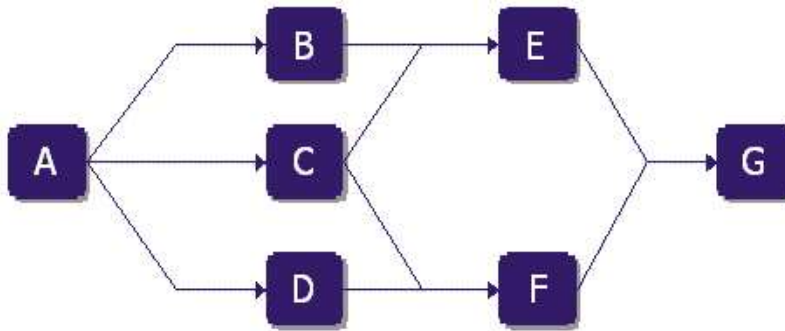


Figure 3: An example of a series-parallel reliability block system

4.4 Formal Programming Methods

Formal programming methods are what the programmers of the “control program”, Coppit and Painter, are relying on to develop a bug-free version of an RBD solver.

The definition of formal methods is as follows: “Mathematically based techniques for the specification, development and verification of software and hardware systems. Referentially transparent languages are amenable to symbolic manipulation allowing program transformation (e.g. changing a clear inefficient specification into an obscure but efficient program) and proof of correctness” [3].

According to Coppit, formal methods are “mathematically precise notations, tools, and techniques used for the development of software. The use of formal methods is widely considered to be expensive, despite evidence that their use can result in software systems with fewer faults. The centerpiece of formal methods is the use of a specification of the software expressed in a mathematically precise notation. The resulting specification is usually more precise, unambiguous, and complete than an informal natural-language specification” [10]. These methods are not usually reproduced by the student programmer and cause the introduction of unexpected bugs, leading to increased debugging time and effort.

Now that we have taken a brief look at the RBD model, formal methods, secure programming, and debugging techniques, we will present the actual tools that attempt to solve the problems of finding bugs and leaks in the RBD solvers, both informally and formally developed.

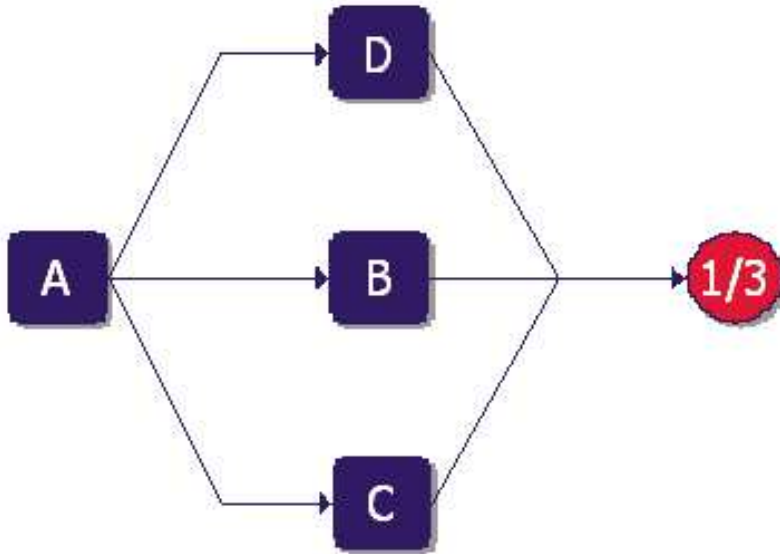


Figure 4: An example of a k-out-of-N reliability block system

4.5 The Tools

Various freely available tools have been chosen for experimentation and evaluation, including Valgrind, Splint, ITS4, ElectricFence, and MemWatc. These debugging tools aim to find the common bugs associated with memory leaks and security issues that take precious time and effort from a programmer on a daily basis. As an introduction to the tools that we will use, a short summary of each tool as well as the bugs that it specializes in finding are listed below. All of the descriptions are derived directly from the tools manual, and in most cases are out to prove that their tool is better than the rest.

4.5.1 Valgrind

Valgrind is a tool that helps find memory-management problems in programs. When a program is run under Valgrind's supervision, all reads and writes of memory are checked and calls to malloc/new/free/delete are intercepted. As a result, Valgrind can detect problems such as: the use of uninitialised memory, reading/writing memory that has been free'd, reading/writing off the end of malloc'd blocks, reading/writing inappropriate areas on the stack, memory leaks (i.e. where pointers to malloc'd blocks are lost forever), passing of uninitialised and/or unaddressible memory to system calls, mismatched use of malloc/new/new[] vs free/delete/delete[], and some misuses of the POSIX pthreads API [1]

Valgrind should work well with finding any memory leaks. We will test this tool for ease of use, as we anticipate its difficulties.

4.5.2 Splint

Splint, an extended version of Lint, is a tool for statically checking C programs for security vulnerabilities and programming mistakes. Splint does many of the traditional Lint checks including unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases. More powerful checks are made possible by additional information given in source code annotations. Annotations are stylized comments that document assumptions about functions, variables, parameters and types (from the Splint website documentation). As more effort is

put into annotating programs, better checking results. Splint is designed to be flexible and allow programmers to select appropriate points on the effort-benefit curve for particular projects. As different checks are turned on and more information is given in code annotations the number of bugs that can be detected increases dramatically.

The problems detected by Splint include: dereferencing a possibly null pointer, using possibly undefined storage or returning storage that is not properly defined, type mismatches, with greater precision and flexibility than provided by C compilers, violations of information hiding, memory management errors including uses of dangling references and memory leaks, dangerous aliasing, modifications and global variable uses that are inconsistent with specified interfaces, problematic control flow such as likely infinite loops, fall through cases or incomplete switches, and suspicious statements, buffer overflow vulnerabilities, dangerous macro implementations or invocations, and Violations of customized naming conventions [7].

Splint seems to have an innumerable amount of features for debugging a C program. We will look for this tool to provide solid bug finding, but we suspect that the ease of use value will not be as high.

4.5.3 ITS4

ITS4 is a tool for statically scanning security-critical C and C++ source code for vulnerabilities. The developers boast that it is efficient enough to give real time feedback to a developer during coding. They also boast extreme ease of use in debugging C++ code. The tool is used to find new remotely exploitable vulnerabilities in software packages, and has become a major piece of e-commerce software [4]. We will determine if the self-testimonials are true and how effective it is at finding any security leaks in the RBDs.

4.5.4 Electric Fence

Electric Fence helps to detect two common programming bugs: software that overruns the boundaries of a malloc() memory allocation, and software that touches a memory allocation that has been released by free(). Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error. It is commonly used with a debugger (e.g., gdb) to detect where such memory violations occur [2], [11]. This tool seems to have a good feedback mechanism and will be helpful in finding memory leaks.

4.5.5 MemWatch

Memwatch, written by Johan Lindh, is an open source memory error detection tool for C. By adding a header file to your code and defining the tool in your gcc statement, you can track memory leaks and corruptions in a C program. Memwatch's strong points are that it provides a log of the results, and detects double-frees, erroneous frees, unfreed memory, overflow and underflow, and so on [5], [11]. We will look for this tool to be effective in finding memory leaks.

5 Summary

Creating a list of effective, freely available debugging tools will benefit all programmers. Knowing that the tools exist and which ones will be most effective in finding a good percentage of their bugs will save time not only in debugging but also in the time that would be spent in researching possible new tools and learning to use them as well. When debugging time is an issue, start-up costs for learning how to debug is even more valuable.

An Analysis of Static Analyzer Tools

Paul DiPalma and Elise Hewett

REFERENCES CITED

- [1] Valgrind, an open-source memory debugger for x86-gnu/linux. URL: <http://developer.kde.org/~sewardj/>.
- [2] Electric fence. URL: .
- [3] Hyperdictionary: Formal methods. URL: <http://www.hyperdictionary.com/computing/formal+methods>.
- [4] Its4: Software security tool. URL: <http://www.cigital.com/its4/>.
- [5] Mastering linux debugging techniques-key strategies to locate and stomp bugs on linux. URL: <http://www-106.ibm.com/developerworks/linux/library/l-debug/>.
- [6] Relex rbd glossary. URL: [url=http://www.relexaustralia.com.au/pages/relex_rbd_glossary.htm](http://www.relexaustralia.com.au/pages/relex_rbd_glossary.htm).
- [7] Splint: Annotation-assisted lightweight static checking secure programming group. URL: <http://www.splint.org>.
- [8] Technical articles and tips: Secure c programming. URL: <http://developers.sun.com/solaris/articles/secure.html>.
- [9] Ross Anderson. Why cryptosystems fail. Cambridge, Massachusetts, November 1993. ACM.
- [10] David Coppit. Career: Runtime verification: Integrating formal methods into common software development practice.
- [11] Crispin Cowan. Software security for open-source systems. Wires Communications, IEEE Computer Society, 2003.
- [12] John Johansen Crispin Cowan, Steve Beattie and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. 12th USENIX Security Symposium, August 2003.
- [13] Freddy's Utilities Plus. Buffer overflows. URL: <http://www.freddys-utilities.co.uk/overflow.php>.