# STACKFENCES: A RUN-TIME APPROACH FOR DETECTING STACK OVERFLOWS

André Zúquete
*IEETA / UA*
*Campus Univ. de Santiago, 3810-193 Aveiro, Portugal*

Keywords: Buffer overflows, run-time detection, run-time correctness assessment, damage containment, dependability

Abstract: This paper describes StackFences, a run-time technique for detecting overflows in local variables in C programs. This technique is different from all others developed so far because it tries to detect explicit overflow occurrences, instead of detecting if a particular stack value, namely a return address, was corrupted because of a stack overflow. Thus, StackFences is useful not only for detecting intrusion attempts but also for checking the run-time robustness of applications. We also conceived different policies for deploying StackFences, allowing a proper balancing between detection accuracy and performance. Effectiveness tests confirmed that all overflows in local variables are detected before causing any severe damage. Performance tests ran with several tools and parameters showed an acceptable performance degradation.

## 1 INTRODUCTION

The exploitation of overflow vulnerabilities has been one of the most popular forms of computer attacks during the last 15 years. According to the ICAT Metabase vulnerability statistics[1], about 20% of the vulnerabilities published in the last 4 years are related with buffer overflows. Such vulnerabilities existed in compiled C or C++ code used for different purposes: operating system (OS) kernel; server and client applications. For some people the problem will continue to exist as long C is used, and can only be minimized or eliminated by using other languages instead of C (McGraw, 2002). However, C is still widely used, and probably will continue to be, because it generates fast compiled code and there is a large repository of legacy code written in C. Therefore, there is a real need for techniques and tools that help to minimize the number and the risk of overflows in old or new C programs.

StackFences is a solution for detecting buffer overflows affecting local variables, i.e., variables allocated in stack frames. The key idea that we explored is an extension to the *canary mechanism*, introduced in StackGuard (Cowan et al., 1998), complemented with the *XOR canary mechanism* introduced also in StackGuard. The original canary mechanisms were deployed for protecting only return values stored in the stack. We extended the protection scope and used canaries for controlling *all stack areas susceptible to overflow*. This way, we hope to detect each and every stack overflow, either affecting highly sensible values, like return addresses, or not. We also conceived different policies for balancing detection accuracy and performance. Namely, run-time validations can be performed in two different ways: (i) one more detailed, allowing a more accurate and timely detection of overflows, more suitable for development scenarios, and (ii) one lazier, checking only when absolutely necessary, more suitable for production environments.

For testing StackFences we developed a prototype for Linux systems using TCC (Tiny C Compiler). C modules compiled with StackFences are fully compatible with the standard C libraries and with modules compiled with other compilers or compilation options.

[1]`http://icat.nist.gov`

## 2 OVERVIEW

There are mainly two reasons for the buffer overflow problems in C programs. First, the language does not check for any boundaries around variables and allows programmers to manage memory areas at will, without any run-time control, using pointers, type casts

Zúquete A. (2004).
STACKFENCES: A RUN-TIME APPROACH FOR DETECTING STACK OVERFLOWS.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 76-84
DOI: 10.5220/0001398000760084
Copyright © SciTePress

and pointer arithmetic. Second, many standard C library functions are intrinsically unsafe concerning buffer overflows (e.g. the infamous `gets()` and several functions for manipulating strings).

Buffer overflows are a problem because they can be used to modify data that controls the execution of a victim process or OS kernel. The exploitation of a buffer overflow vulnerability can expose a victim application to two different risks:

**Denial of Service (DoS):** an attacker may interfere randomly with the application's execution flow, eventually making it fail after some illegal operation.

**Penetration:** An attacker may take control of the application's execution flow by performing a crafty buffer overflow.

Ideally, one would like to avoid both risks, but that may be difficult or even impossible to achieve with current hardware architectures and existing software. Comparing both risks, the risk of penetration is greater than the risk of DoS. Therefore, it seems useful to avoid penetration risks by transforming them into DoS risks. We followed this reasoning in the design of StackFences.

## 2.1 Detection vs. prevention

Buffer overflows may be detected using static analysis, i.e., before actually compiling and deploying the code (e.g. (Wagner et al., 2000; Larochelle and Evans, 2001)). Another possibility is to tackle buffer overflows dynamically, or at run-time, during the execution of the vulnerable applications. This was the approach that we followed in StackFences.

Dealing with buffer overflows at run-time implies either prevention or detection. *Prevention* attempts to conceal overflow vulnerabilities or to make their occurrence partially or totally harmless. Though prevention does not help finding problems out, it is highly desirable for avoiding both DoS and penetration risks. However, total prevention is difficult to achieve.

*Detection* attempts to detect the occurrence of abnormal facts, such as buffer overflows. Detection only mitigates the problem, since the usual reaction is to raise some sort of alert and immediately terminate the affected application or OS kernel, eventually leading to a DoS situation. Therefore, detection helps to assess correctness on the execution of applications with overflow vulnerabilities, which is important for improving damage containment and reducing penetration risks. StackFences is a detection solution.

## 2.2 Anatomy of a stack overflow

The overflow of the stack memory reserved for a C variable can corrupt the execution flow of an application in many ways. Assuming only overflows across memory with growing addresses and an x86 microprocessor, we have at least the following scenarios:

1. By setting the value of a neighbouring variable below in the stack.
2. By setting the value of a saved frame pointer of a previous stack frame to another address. When the modified frame pointer is recovered, the application will use, in that stack frame, different local variables and function arguments.
3. By setting the return address for the calling function. When the function returns it will continue to execute at an address chosen by the attacker.

These attacks can be complemented by inserting auxiliary data, like microprocessor instructions or function parameters, into input buffers, overflowed or not. The auxiliary data can be used later for a penetration bootstrap if the attacker succeeds in using it.

Most overflow attacks are *stack-smashing attacks* (Aleph One, 1996), i.e., attacks of the third type referred above, that overwrite return addresses and jump into bootstrap penetration code. The famous Internet Worm of 1988 did it (Spafford, 1989), as well as many other attacks thereafter.

But the two other types are also risky. J. Wilander and M. Kambar pointed out, in (Wilander and Kambar, 2002), that a general weakness of the solutions presented so far for run-time detection or prevention is that they protect *known attack targets*, mainly return addresses in the stack, instead of protecting *all targets*. From a pragmatic point of view, we can understand why that happens: stack-smashing attacks are the simplest and most popular ones. But for dealing with future and more sophisticated attacks exploiting buffer overflows we need to change the protection paradigm, like we did with StackFences.

## 3 RELATED WORK

In this section we describe the approach followed by several run-time overflow detection or prevention techniques. We do not address any static detection solutions.

Many run-time protection techniques were developed to protect the most common target of overflow attacks: the return pointer. StackGuard (Cowan et al., 1998) uses *canaries*, which are specific values that are placed between the local variables and the return address in the same stack frame. The canary is installed in the function prologue and checked in the epilogue. If its value changed in the meanwhile then there was an overflow and the process is halted; otherwise the function returns normally.

`Propolice` (Etoh and Yoda, 2000) enhances the basic protection of StackGuard by rearranging local variables. The assumptions of `propolice` are that (i) only character arrays are vulnerable to overflows and (ii) function arguments do not contain character arrays. Thus, vulnerable local variables — character arrays or structures with character arrays — are packed together in a *vulnerable location* next to the canary (*guard*). All other variables are placed above in the stack. This way, overflows can occur inside vulnerable locations but cannot affect non-vulnerable variables neither the return address, which is protected by the canary. However, `propolice` assumptions are not complete, because there are other kinds of vulnerable variables besides character arrays, and ignores overflows affecting only variables in a vulnerable area.

Another way of protecting return addresses is by hiding them with a *XOR canary*, or *cookie*. The XOR canary is XORed with the return address in a function's prologue and again in the epilogue. Attackers not knowing the value of the XOR canary are unable to modify return addresses in a useful manner. StackGuard was the first to use XOR canaries to frustrate attacks (to return addresses) circumventing the basic canary mechanism[2]. StackGhost (Frantzen and Shuey, 2001) is kernel-level solution for Sparc architectures that protects return addresses with XOR canaries. It uses either per-kernel or per-process XOR canaries, but the first is too weak for competent hackers. Overflow attacks affecting return values produce wrong return addresses. These can be automatically detected in 75% of the cases because Sparc instructions must be aligned on a 4-byte boundary; on the other 25% the program will run uncontrolled.

StackGuard's MemGuard (Cowan et al., 1998) protection makes return addresses in the stack read-only during the normal execution of functions. This way, any attempts to overwrite them raise a memory exception. However, the performance penalty of this approach is huge.

Another way of protecting return addresses is to keep a separate copy of them in a *return-address stack*. Return addresses are stored in and fetched from the return-address stack in the function's prologue and epilogue, respectively. Vendicator's StackShield[3] `Global Ret Stack` protection and Secure Return Address Stack (Xu et al., 2002) use only the copied values, thus preventing attacks affecting return addresses stored in the normal stack. But these solutions fail completely in protecting any other stack values from overflows.

The Return Address Defender (Chiueh and Hsu, 2001) also uses a return-address stack but provides only detection because return addresses on the Return Address Repository are compared with the ones in the ordinary stack before being used, and the process is halted if they are different. StackShield's `Ret Range Check` is similar but stores a copy of the current return address in a global variable. They are both useless against overflow attacks overwriting the return address with its exact value, which is not difficult to guess for a competent hacker.

Libverify (Baratloo et al., 2000) also uses a return-address stack but it can be transparently applied to existing binary code by means of a dynamic library. The drawback of Libverify is that all protected code must be copied into the heap to overwrite instructions in the prologue and epilogue of all functions. This means that processes are unable to share the code they effectively run in main memory and absolute jumps within the text area must be handled with traps.

Libsafe (Baratloo et al., 2000) is a dynamic library that replaces unsafe functions of the standard C library that are typically used for performing buffer overflows. All Libsafe functions compute the upper bounds of destination's buffers before actually transferring data into memory. The upper bounds are defined by the location of return addresses. But the protection is limited and misses unsafe functions compiled inline within existing applications or libraries.

PointGuard (Cowan et al., 2003) is an extension of the original XOR canary mechanism for protecting pointer variables. Pointers are stored encrypted (XORed with the XOR canary) and are decrypted when loaded into CPU registers. However, this approach raises problems when integrating mix-mode code (some PointGuard, some not).

Practically all protection mechanisms look only at the effects of overflows within the current stack frame — exceptions are PointGuard and Libsafe. StackGuard 3 (Wagle and Cowan, 2003) and Libsafe also protect saved frame pointers, which are collocated with return addresses. And `propolice`, *under their assumptions*, further protects all variables that are not character arrays or structures with character arrays.

Most of the protection mechanisms described are added at compile time; exceptions are Libsafe, Libverify and StackGhost. StackFences is also added at compile time.

With StackFences we tried to further improve the detection of stack overflows. Namely, we tried to (i) detect overflows within all existing stack frames, not only the current one; and to (ii) detect overflows from all variables that could be overflowed. In this particular case we extended `propolice`'s and PointGuard's notions of vulnerable areas and we do not ignore overflows within variables belonging to a vulnerable area, as `propolice` does.

[2]This mechanism was introduced in version 1.12.

[3]`http://www.angelfire.com/sk/stackshield`

## 4 OUR CONTRIBUTION: StackFences

To detect overflows in stack variables we use StackGuard-like boundary canaries between them. The management of canaries — allocation, set-up and checking — depends on the kind of stack variables we are dealing with: local variables or function parameters. In this paper we handle only the management of boundary canaries for local variables.

The value of canaries is protected with a *XOR canary*, that is a per-process random 32-bit value. The canary is stored in a publicly known variable (`cXor` hereafter) and should not be modified after being setup. Adaptive attacks trying to guess the correct value of a process' canary are infeasible, in theory, because attacked processes should terminate after an attack with an unsuitable, tentative canary value. We assume, like for StackGuard, StackGhost and PointGuard, that attackers cannot get dumps of memory areas containing any well-known values XORed with the XOR canary.

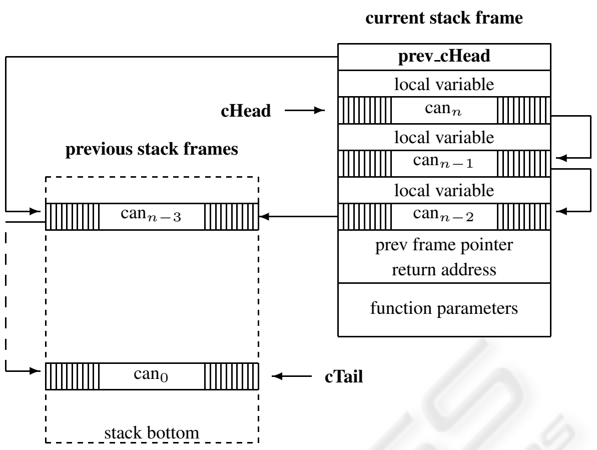

Figure 1: Example of the list of canaries, starting in the current stack frame ($\text{can}_n$) and until the first one inserted at the beginning of the process execution ($\text{can}_0$). Variables `cHead` and cTail point the head and tail of the list. Shaded boxes represent canaries XORed with the process' XOR canary.

### 4.1 Overview

Boundary canaries are located near local variables, on higher addresses. The questions now are (i) which variables we want to, or should, bind with canaries, (ii) how do we find the canaries and check their value; and (iii) when do we want to, or should, check the value of canaries.

#### 4.1.1 Variables to bind with canaries

Following a high security policy, we should bind all stack variables. But this is a sort of brute force approach that has considerable impact in the performance of modified applications.

A more relaxed policy, which we chose for StackFences, is to bind only *potentially vulnerable variables*. These are the variables *for which a pointer is taken and used in the current function or other function called upon it*. This policy is more efficiency than the previous one and should tackle most stack overflow problems, since they usually derive from deficient uses of legitimate stack addresses.

Because the size of C structures cannot be altered, canaries cannot be added between members of local `struct` variables. Therefore, overflows may still occur inside structures, but not affecting external memory areas.

#### 4.1.2 Looking for and checking canaries

We used a function-independent way of locating and checking canaries, because it appeared to be more flexible and simple to implement and test. The set of all boundary canaries forms a linked list, as shown in Fig. 1. The location of the head and tail canaries of the list are stored in publicly known variables (`cHead` and `cTail` in this text). The virtual address (of another canary) stored in each canary is protected using the XOR canary previously referred. This way an attacker causing the overflow of a stack variable cannot easily guess valid values for boundary canaries between the overflowed variable and the neighbouring variables to be tampered.

The full list of canaries, or particular sub-lists, can easily be checked by dereferencing canaries, XORing the obtained value with the XOR canary and testing whether the result is a valid address. Testing the validity of an address is straightforward: (i) it must be higher than the previous one, because the list goes strictly from the top to the bottom of the stack, and (ii) it cannot be higher than a target canary address that we want to reach. Any violation of these assertions is an overflow evidence.

#### 4.1.3 Triggering canary checking

There are at least two distinct situations that should trigger canary checking:

**Before doing some operation related with the external perception of the application's behaviour**. Broadly, this means that checking should be done before any I/O attempt;

**Just before the return of a function**. In this case, we should check for overflows within the current stack frame, i.e., caused in local variables or parameters, which will disappear because the stack frame will be released.

In the first case, we should check the full list of canaries, from `cHead` to `cTail`, because we are looking for any stack overflow. The more times it is checked, the more timely we can find stack overflows, but with a significant impact on the performance of the application.

In the second case, we should check only the list of canaries belonging to the local stack frame ($\texttt{can}_n$ to $\texttt{can}_{n-2}$ in Fig. 1) since we are looking for evidences of local stack overflows that are about to disappear. If no canaries exist in the current stack frame then no local checking is required. Checking a local list of canaries is also an iterative walk starting in `cHead` but ending in the head canary of the previous stack frames ($\texttt{can}_{n-3}$ in Fig. 1).

## 4.2 Management of the canary list

StackFences' canaries are similar to local variables, but are invisible to application code and have a compiler-defined (initial) value. The stack space for canaries is allocated when the compiler defines the location of local variables. The setup of the canaries should occur after the normal C prologue.

The list of canaries is increased after the prologue of a function and decreased when the function returns. Increasing the list consists of adding all canaries in the local stack frame to the head of the list and storing in `cHead` the address of the new top-most stack canary ($\texttt{can}_n$ in Fig. 1). Decreasing the list consists simply of setting the value of `cHead` using the address of the head canary of the previous stack frames ($\texttt{can}_{n-3}$ in Fig. 1).

The list of canaries must also be increased when the program calls `alloca`. The space requested should be increased to accommodate a canary at the end of it, which becomes the new list head. The list must also be decreased when the program calls `longjmp`; in this context it is similar to a *long return*. The value of `cHead` must be set with the address of the head canary in the stack frame we are jumping into, or in some other stack frame below.

## 4.3 Policies for canary checking

Checking boundary canaries is a potentially expensive operation, thus it needs to be carefully managed in order to balance two requirements: (i) effective detection of stack overflows and (ii) efficient execution of the program. Furthermore, in terms of effectiveness, two natural approaches should be contemplated considering the software life cycle: (i) in development stages, the sooner overflows are spotted the better, while (ii) in production stages, preventing applications from "making damage" may be enough for most cases.

Thus, considering the two execution environments mentioned above – development and production – we conceived two different policies for checking the correctness of boundary canaries. The two policies define when two lists of canaries are checked: (i) the list of canaries belonging to the current stack frame and (ii) the full list of canaries.

### 4.3.1 Checking local canaries

As previously explained, boundary canaries on the current stack frame should be checked when the stack frame is about to be released; otherwise, we could miss local overflows. Consequently, the reduction of the canary list, both within a normal function return and within a call to `longjmp`, always checks the consistency of all released canaries.

The extra code for checking local canaries and reducing the list of canaries was placed in a function (`canReduce`) that is called before the function's epilogue. It gets, as a parameter, the address of the head canary in the previous stack frames ($\texttt{can}_{n-3}$ in Fig. 1). This address is stored (in clear) at the top of the current stack frame in the function's prologue (`prev_cHead` in Fig. 1). The function `canReduce` checks the canaries from `cHead` until `prev_cHead` and, at the end, stores `prev_cHead` in `cHead`.

For handling the reduction of the canary list after a `longjmp` call we use a function similar to `canReduce`. The function starts from `cHead` and walks along the canary list until finding a canary with an address higher than the stack pointer saved in the jump context. The address of that canary will be stored in `cHead`. Jump contexts stored by `setjmp` and used by `longjmp` are not modified; only the functionality of `longjmp` needs to be extended.

### 4.3.2 Checking all canaries

We conceived two policies for checking the full list of canaries: one more suitable for development scenarios, another more suitable for production environments. In either case, we tried to prevent an attacked process from doing any I/O after a stack buffer overflow. The checking function is named `canWalk`.

**Development policy:** In a development scenario we want to catch an overflow as accurately as possible, in order to simplify the process of finding and fixing the vulnerability. It is, thus, natural to sacrifice execution efficiency in favour of debugging effectiveness. Our development policy consists of a `canWalk` call before each function call. This approach is computationally costly but has the advantage of detecting overflows not far from where they occurred.

**Production policy:** In a production scenario we want the applications to run efficiently and, yet, we want to detect overflows before they can interfere with their I/O. Our production policy consists of a watchdog process to catch all the system calls of the target process and to call `canWalk` before each I/O system call requested by the target process. We define an I/O system call as a system call interacting with I/O objects (files, pipes, sockets, etc.) or with other operating system resources (e.g. send signals to other processes, manage virtual memory attributes, change the ownership of the process, etc.).

## 5 IMPLEMENTATION

For implementing all the mechanisms and policies previously described for detecting stack buffer overflows we modified a C compiler and developed 3 auxiliary C modules for Linux systems that must be linked with the applications we want to protect. The applications only need to be recompiled with the modified compiler and linked with some of the modules to use StackFences.

For the C compiler we used the version 0.9.20 of TCC (Tiny C Compiler), a simple but complete C compiler developed by Fabrice Bellard (Bellard, 2003). We chose this compiler mainly because it allows a fast prototyping for getting a proof of concept.

The first auxiliary module defines variables and functions that are common to all overflow checking facilities — the public variables `cXor`, `cHead` and `cTail`; the XOR canary setup function (that uses the file `/dev/urandom`), the function `canReduce`; and a new `longjmp` function.

The second auxiliary module defines a new `main` and all the checking/aborting functions for the development policy. The new `main` defines the initial boundary canary, initiates variables in the previous module and calls the application's original `main`.

The third auxiliary module defines a new `main` and all the checking/aborting functions for the production policy. The new `main` creates a child process for running the application and the initial process stays as the watchdog of the new one (using `ptrace`). The code of new `main` executing in the child process is basically identical to the one of the second module.

The watchdog triggers a call to `canWalk` before allowing the execution of any requested I/O system call. The `canWalk` function is similar to the one in the second module but fetches canary values with `ptrace` because it runs in a different process.

## 6 EFFECTIVENESS EVALUATION

We tested the effectiveness of StackFences with the test suite described in (Wilander and Kambar, 2002) and kindly provided by the authors. The results were the best possible: StackFences detected and halted *all* 12 attacks overflowing stack variables. We were also able to do so using either of the canary checking policies described in §4.3.2.

The empirical results obtained with the same test suite and using 4 protection tools — StackGuard, StackShield, `propolice` and Libsafe/Libverify – showed that the tools where able to handle, in the best case, 10 out of the 12 attacks (with `propolice`). Note, however, that:

- the test suite is not complete, it only overflows character arrays, which are exactly the vulnerable variables considered by `propolice`. But StackFences is more powerful, being able to detect overflows in local variables other than character arrays. Therefore, StackFences is much better suited for assessing the correctness of applications in runtime than `propolice`, but that cannot be fully demonstrated with this test suite.
- `propolice` does not detect any overflows within consecutive character arrays, as StackFences does, and such vulnerability is also not explored by the test suite.

## 7 PERFORMANCE EVALUATION

The performance penalties introduced by StackFences depend on several factors, namely: (i) the number of vulnerable local variables; (ii) the number of function calls; (iii) the length of the list of canaries when `canWalk` is called; and (iv) the number of I/O operations that trigger the call to `canWalk` (if using the production policy).

We evaluated StackFences with both micro and macro benchmarks. The micro benchmarks provide upper bounds to the overheads caused by setting and checking canaries in each function's prologue and epilogue, respectively. The macro benchmarks help us to have an idea about the relative cost of each checking policy, because they control the calls to `canWalk`, and also to get an idea about the space occupied by canaries. But, for lack of space, micro benchmarks are not further referred in this article.

For the evaluation of StackFences with macro benchmarks we chose 3 tools with moderate file I/O and high CPU activity: `ctags`, `tcc` and `bzip2`. These 3 tools where compiled in 4 different ways — with `gcc`, `tcc`, and `tcc` with the two StackFences

Table 1: Results of the execution of macro benchmarks with existing applications for evaluating the total overheads introduced by StackFences and statistic data regarding the checking of StackFences' canaries.

| | | | | | elapsed time (ms) | | | | StackFences statistics | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tool | local variables | | arguments | StackFences | gcc | tcc | | | canReduce | | | canWalk | | |
| | total | vulnerable | | policy | -O3 | orig. | with | | calls | length | | calls | length | |
| | | | | | | | StackFences | | | avg. | max. | | avg. | max. |
| | | | bzip2 | development | 70 | 122 | 149 | (+22%) | 1,667 | 5.9 | 11 | 437,946 | 21.8 | 24 |
| | | | sources | production | | | 126 | (+3%) | | | | 48 | 7.4 | 13 |
| bzip2 | 462 | 302 | tcc | development | 207 | 324 | 396 | (+22%) | 1,970 | 6.0 | 11 | 1,127,521 | 22.1 | 24 |
| | | (65%) | sources | production | | | 334 | (+3%) | | | | 62 | 8.4 | 13 |
| | | | ctags | development | 230 | 360 | 439 | (+22%) | 1,683 | 5.9 | 11 | 1,207,621 | 22.3 | 24 |
| | | | sources | production | | | 371 | (+3%) | | | | 60 | 8.5 | 13 |
| | | | bzip2 | development | 89 | 167 | 419 | (+151%) | 411,256 | 2.4 | 10 | 2,927,026 | 30.8 | 165 |
| | | | sources | production | | | 217 | (+30%) | | | | 1,250 | 14.6 | 65 |
| tcc | 978 | 603 | tcc | development | 97 | 179 | 517 | (+189%) | 464,208 | 2.5 | 10 | 3,737,023 | 33.5 | 252 |
| | | (62%) | sources | production | | | 222 | (+24%) | | | | 928 | 15.9 | 60 |
| | | | ctags | development | 304 | 541 | 1,149 | (+112%) | 1,231,803 | 2.3 | 10 | 8,297,317 | 24.1 | 163 |
| | | | sources | production | | | 724 | (+34%) | | | | 4,682 | 13.9 | 102 |
| | | | bzip2 | development | 36 | 36 | 49 | (+36%) | 3,338 | 1.4 | 3 | 1,334,837 | 2.1 | 5 |
| | | | sources | production | | | 44 | (+22%) | | | | 137 | 2.3 | 4 |
| ctags | 911 | 178 | tcc | development | 91 | 92 | 132 | (+43%) | 13,899 | 1.4 | 3 | 3,599,858 | 2.2 | 5 |
| | | (20%) | sources | production | | | 116 | (+26%) | | | | 207 | 2.3 | 4 |
| | | | ctags | development | 91 | 98 | 138 | (+41%) | 16,530 | 1.4 | 3 | 3,480,356 | 2.2 | 5 |
| | | | sources | production | | | 138 | (+41%) | | | | 391 | 2.2 | 4 |

checking policies — and executed with 3 different parameters — the sources of each of the 3 tools. All benchmarks ran in a Red Hat 8.0 Linux box (kernel 2.4.18-27.8.0) with a Pentium IV at 2.4 GHz, 256 MB RAM and 512 KB cache.

The results of the evaluation are presented in Table 1: the second and third columns show the number of local variables used by the tools and the number and percentage of them that are vulnerable and checked by StackFences; the fourth column shows the arguments used with the tools; the fifth column shows the protection policy used with StackFences. In the rest of the columns we have execution results: the sixth and seventh columns show the elapsed time observed with the tools compiled with `gcc` (with maximum optimisation) and the normal `tcc` (that has no optimisations); the eighth column shows the elapsed time observed with the tools compiled with `tcc` and StackFences, using both security policies, and the overhead in percentage comparing with the results of the previous column; the last six columns show the number of calls to `canReduce` and `canWalk` and the average and maximum number of canaries checked per call. StackFences' auxiliary modules were written in C and compiled with `gcc` and maximum optimisation. The elapsed times are the minimum observed in 100 consecutive runs of each test.

The values provided for `gcc` are only indicative, because many dynamic solutions for dealing with buffer overflows were implemented with it. But it doesn't make sense to extrapolate the overhead of StackFences comparing with `gcc` because some of the optimisations used by the latter would also reduce the overhead of StackFences if it was part of `gcc`. For instance, `tcc` has many small `inline` functions. Since our version of TCC ignores `inline` qualifiers, that greatly increases the number of `canReduce` calls and the average number of canaries checked by `canWalk`. Such overhead would not happen if we had integrated StackFences with `gcc`.

The results in Table 1 show that the overheads introduced by StackFences are acceptable. With the development policy, overheads are between 22% and 189% of the elapsed time with `tcc` and without StackFences. With the production policy, overheads are lower, as desired, between 3% and 41%. The production policy greatly reduces the number of calls to `canWalk`, as expected. But in some cases, namely with `ctags`, results show that it is almost irrelevant to use either checking policy. That happens because the average number of canaries checked by `canWalk` is low, making more relevant the cost of the process switching between the application and its watchdog in the production policy.

Considering the applications tested, the extra space occupied by canaries in the stack is not an issue. In the worst case there is a maximum of 252+10 canaries (when `tcc` is compiled by itself using the development policy), which represents a memory overhead of about 1 KB.

A small comparison can be established between the performance of `ctags` with StackGuard and with

StackFences. According to (Cowan et al., 1998), `ctags` with StackGuard has an overhead of 80% when processing 78 files, 37,000 lines of code. Using the same number of files and an approximated number of lines of code (37,188) we got for StackFences a lower performance penalty: 31% with the production policy and 41% with the debug policy.

According to Table 1, `gcc` and `tcc` generate equally fast `ctags` executables. Therefore, for this particular experience we can compare the overheads of the two protection mechanisms independently of the compilers implementing them. And the conclusion is that StackFences is faster than StackGuard, which makes less validation actions! This paradox is probably explained by the fact that the performance figures presented in (Cowan et al., 1998) were obtained with a *non-optimized version of StackGuard, that added canaries and code for checking them to all functions*, instead of doing it only for functions with vulnerable local variables. This fact is mentioned in the performance optimisations for StackGuard described in (Cowan et al., 1998). StackFences, on the contrary, was optimized to add canaries and for checking them locally with `canReduce` *only in functions with vulnerable variables*.

## 8 CONCLUSIONS

We have presented StackFences, a run-time solution for detecting buffer overflows affecting variables allocated in stack frames. StackFences detects overflows in all potentially vulnerable local variables instead of detecting only overflows affecting known attack targets, like return addresses. To the best of our knowledge this is the first solution to perform such a detailed run-time stack analysis.

For balancing detection accuracy and performance we conceived two checking policies for StackFences: a development policy, more detailed and allowing a more accurate and timely detection of overflow occurrences, suitable for development scenarios; and a production policy, lazier, checking only when absolutely necessary, suitable for production environments.

In terms of effectiveness, StackFences detected and halted *all* the 12 attacks of the test suite described in (Wilander and Kambar, 2002). Although it may appear that StackFences is minutely different from `propolice`, which avoids 10 of those attacks, that is not true because StackFences detects other overflows that are not considered by the test suite and neither tackled by `propolice`. StackFences is much better suited for assessing the correctness of applications in run-time than `propolice`, though the test suite does not properly demonstrate that.

The performance of StackFences is acceptable but depends a lot on the compiled application. We believe that optimizing canary checking code can further reduce StackFences' overheads. As desired, the overhead of the production policy is lower than the overhead of the debug policy. Finally, StackFences' overheads are mainly compiler independent, though they may depend on some compiling options.

## REFERENCES

Aleph One (1996). Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49).

Baratloo, A., Singh, N., and Tsai, T. (2000). Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, USA.

Bellard, F. (2003). Tiny C Compiler, version 0.9.20. `http://fabrice.bellard.free.fr/tcc`.

Chiueh, T.-C. and Hsu, F.-H. (2001). RAD: A Compile-time Solution to Buffer Overflow Attacks. In *IEEE Int. Conf. on Distr. Computing Systems (ICDCS)*, Phoenix, AZ, USA.

Cowan, C., Beattie, S., Johansen, J., and Wagle, P. (2003). PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *12th USENIX Security Symp.*, Washington, D.C., USA.

Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. (1998). StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conf.*, pages 63–78, San Antonio, TX, USA.

Etoh, H. and Yoda, K. (2000). propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer SECurity Abstract*, 43(12). `http://www.ipsj.or.jp/members/Journal/Eng/4312/article053.html`.

Frantzen, M. and Shuey, M. (2001). StackGhost: Hardware Facilitated Stack Protection. In *Proc. of the 10th USENIX Security Symp.*, Washington, D.C., USA.

Larochelle, D. and Evans, D. (2001). Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of the 10th USENIX Security Symp.*, pages 177–190, Washington, D.C., USA.

McGraw, G. (2002). Building Secure Software. In *RTO/NATO Real-Time Intrusion Detection Symp.*, Estoril, Portugal. Invited Talk.

Spafford, E. H. (1989). The Internet Worm Incident. In Ghezzi, C. and McDermid, J. A., editors, *ESEC'89 2nd European Software Engineering Conf.*, University of Warwick, Coventry, United Kingdom. Springer.

Wagle, P. and Cowan, C. (2003). StackGuard: Simple Stack Smash Protection for GCC. In *Proc. of the GCC Developers Summit*, pages 243–255.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A First Step towards Automated Detection of

Buffer Overrun Vulnerabilities. In *Proc. of the Internet Soc. Symp. on Network and Distr. Systems Security (NDSS 00)*, pages 3–17, San Diego, CA, USA.

Wilander, J. and Kambar, M. (2002). A Comparison of Publicly Available Tools for static intrusion prevention. In *Proc. of the 7th Nordic Workshop on Secure IT Systems*, pages 68–84, Karlstad, Sweden.

Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K. (2002). Architecture Support for Defending Against Buffer Overflow Attacks. In *2nd Works. on Evaluating and Architecting System Dependability (EASY)*, San Jose, CA, USA.