StackFences: a run-time approach for detecting stack overflows

André Zúquete

Abstract – **This article 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. For testing StackFences we developed a prototype for Linux systems using TCC (Tiny C Compiler). C modules compiled with StackFences are fully compatible with C modules compiled differently and standard libraries. 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.**

Resumo – **Este artigo descreve o StackFences, uma técnica para detectar em tempo de execução transbordamentos de memória em variáveis locais de programas em C. Esta técnica é diferente das demais desenvolvidas para lidar com este problema porque detecta directamente os transbordamentos de memória, em vez de detectar se valor específicos na pilha, como endereços de retorno, foram corrompidos devido a um transbordamento de memória. Assim, o StackFences é útil não só para detectar tentativas de intrusão mas também para monitorizar a correcção de execução das aplicações. Foram também concebidas duas políticas de exploração do StackFences que permitem um equilíbrio apropriado entre correcção e desempenho. Para testar o StackFences desenvolveu-se um protótipo para sistemas Linux usando o TCC (Tiny C Compiler). Os módulos C compilados com o StackFences são totalmente compatíveis com módulos C compilados diferentemente ou com bibliotecas padrão. Os testes de eficácia confirmaram que todos os transbordamentos em variáveis locais são detectados antes de causar um estrago significativo. Os testes de desempenho executados com diversas ferramentas e parâmetros revelaram uma degradação de desempenho aceitável.**

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

Palavras chave – **Transbordamentos de memória, detecção em tempo de execução, verificação de correcção em tempo de execução, minimização de estragos, confiança operacional**

This article is an extended version of another one published in the 1st International Conference on E-business and Telecommunication Networks (ICETE 2004).

I. 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 [1]. 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.

In this paper we address only problems caused by *buffer overflows*, i.e., with write operations outside the proper memory area, invading neighboring memory areas. But there are other kinds of overflows, such as overflows in the values of variables. These overflows do not directly affect neighboring memory areas but do modify the execution flow of the application (e.g. the SSH daemon integer-overflow vulnerabilities CVE-2001-0144 and CVE-2002-0639 reported in the ICAT Metabase).

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 [2], 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), a very simple, though complete C compiler. The code generated by our modified version of TCC manages boundary canaries near local variables susceptible to overflow. C modules

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

compiled with StackFences are fully compatible with the standard C libraries and with modules compiled with other compilers or compilation options. The modifications introduced by StackFences affect only the code inside each function, and the external visibility of such modifications is required only for some StackFences' checking procedures.

This paper is structured as follows. Section II overviews the issues raised by stack buffer overflows. Section III presents the related work concerning run-time detection/prevention of stack buffer overflows. Section IV presents our contribution to detect stack buffer overflows. Section V presents some implementation details regarding the modified compiler and some extra modules with auxiliary variables and functions. Sections VI and VII evaluate the effectiveness of StackFences and the overheads introduced in the execution of microbenchmarks and real applications. Finally, Section VIII presents the conclusions and future work.

II. 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 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 [3]).

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. The damage caused by such an attack is difficult to assert, both for the attacker and the victim, because the attacked application may fail without any control.

Penetration: An attacker may take control of the application's execution flow by performing a crafty buffer overflow. In principle, the attacker knows very well the damage produced by the attack, or produced by conducting an intrusion thereafter.

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.

A. *Detection/prevention of buffer overflows*

Buffer overflows may be detected using static analysis, i.e., before actually compiling and deploying the code (e.g. [4], [5]). This approach, however, is not likely to be complete, even though it may be used to spot well-known vulnerabilities. Another possibility is to tackle buffer overflows dynamically, or at run-time, during the execution of the vulnerable applications. In this case, the vulnerable application, or the execution environment, must be instrumented to prevent or detect overflows or problems caused by overflows. This was the approach that we followed in StackFences.

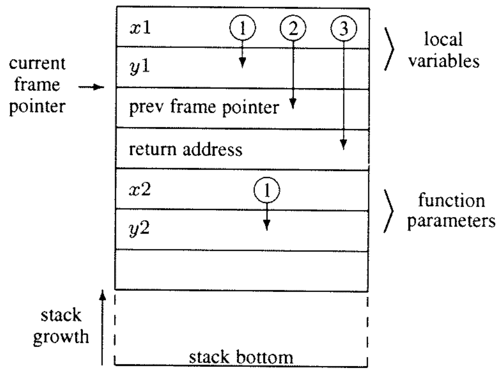


Figure 1 - Possible overflows of the stack memory reserved for a C variable x (either $x1$ or $x2$). Overruns may affect (1) neighboring variables $y1$ or $y2$, (2) a saved frame pointer, or (3) a return address.

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. If the prevention is effective a vulnerable application may continue to execute normally even when under attack. Though prevention does not help finding problems out, it is highly desirable for avoiding both DoS and penetration risks. But total prevention is difficult to achieve and partial prevention should be avoided because it gives a false sense of security.

Detection attempts to detect the occurrence of buffer overflows or the occurrence of abnormal facts that may be consequence of 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 or OS kernels with overflow vulnerabilities, which is important for improving damage containment and reducing penetration risks. StackFences is a detection solution.

B. *Anatomy of a stack overflow*

The overflow of the stack memory reserved for a C variable x can corrupt the execution flow of an application in many ways. Assuming only overflows across memory with growing addresses and a x86 processor, we have at least the following scenarios (cf. Fig. 1):

1. By setting the value of a neighboring variable y. If y is a pointer to a function, one can reassign it to a different function. If y is a pointer to a memory area, one can cause a (probably fixed) memory modification somewhere else. If y is a `longjmp` buffer, one can control a future jump using it. If y is an integer, float or struct/union variable one can modify the flow of the program if the value of y is relevant for making flow

decisions or producing useful results.

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, since the frame pointer is the base address for addressing them.
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* [6], 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 [7], as well as many other attacks thereafter. But the two other types are also risky. For instance, M. Rolf presents in [8] a difficult, though possible, exploit using a tampered frame pointer. Furthermore, the use of *canaries* and other protection mechanisms for detecting the modification of return addresses (cf. Section III) pushes attackers to work around them in order to exploit stack buffer overflows without triggering those protection mechanisms. Thus, by detecting as much as possible all stack overflows we should be able to defeat more overflow attacks, both already known or still unknown.

C. Protection paradigms

J. Wilander and M. Kambar pointed out, in [9], 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.

In [9] it is also mentioned that *"changing the flow of control occurs by altering a code pointer"*. This is true but not complete. Namely, we can change the flow of control by overflowing a simple integer variable. It may be argued that it should be difficult to succeed in a penetration attempt using such a simple overflow vulnerability, but the fact of not knowing any instance of such an attack doesn't mean that it could not appear in the future. And, in any case, such apparently useless overflows may lead to DoS situations, either by crashing the victim application or, even worse, by leading it to an abnormal behavior. Thus, for ensuring the run-time correctness of an application we should control each and every variable susceptible to be affected by an overflow; we tried to do so for local variables with StackFences.

III. 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 [2] 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. Two types of canaries where proposed for StackGuard — *terminator canary* for preventing overflows with character strings and *random canary* computed in run-time — but only the first is actually used [9].

`Propolice` [10] 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 after the vulnerable location, above in the stack. This way, overflows can occur inside vulnerable locations but cannot affect non-vulnerable variables, because they are in lower stack addrtess, neither the return address because it 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 [11] is kernel-level solution for Sparc architectures that protects return addresses this way. StackGhost 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 [2] 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.

[2]This mechanism was introduced in version 1.12.

Vendicator's StackShield[3] `Global Ret Stack` protection and Secure Return Address Stack [12] use only the copied values, thus preventing attacks affecting return addresses stored in the normal stack. These solution are very good in keeping return addresses correct but fail completely in protecting any other stack values from overflows.

The Return Address Defender [13] 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. All these detection mechanisms are useless against overflow attacks overwriting the return address with its exact value, which is not difficult to guess for a competent hacker.

Libverify [14] is another protection mechanism that uses a return-address stack but, unlike the former, 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.

Yet another way of protecting return addresses is to check memory limits within certain critical functions. For instance, Libsafe [14] 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 [15] 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. The solutions like StackGuard, that use a boundary canary, can also protect frame pointers stored next to return addresses; it is only necessary to place the canary between the frame pointer and the local variables. StackGuard 3 [16] and Libsafe protect frame pointers. 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.

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

IV. OUR CONTRIBUTION: STACKFENCES

To detect overflows in stack variables we decided in favor of managing StackGuard-like boundary canaries between them. The management of canaries, which involves their allocation, setup 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. But a similar approach for function parameters is already scheduled for future work.

To protect the value of canaries we use a *XOR canary*. Like in StackGhost and PointGuard, this is a per-process random 32-bit value that is used to hide important values. It can be compared to a 32-bit long keystream that is used to encrypt and decrypt sensitive values. The canary is stored in a publicly known variable (`cXor` in this text) 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 also assume, like for StackGhost and PointGuard, that it is impossible for an attacker to get dumps of stack areas containing any boundary canaries XORed with the XOR canary.

As for StackGuard, we assumed that overflows are caused by writing continuous amounts of data from a memory address pointing to any byte of the correct memory area. But, unlike StackGuard, we can also tackle certain overflows caused from wrong pointers, i.e., pointers referring to (semantically) wrong memory areas. For instance, such pointers appear when wrong indexes, both positive or negative, are used to access arrays.

A. Overview

Boundary canaries are located near local variables, on higher addresses. The questions now are (i) which variables we want to, or should, bound 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.

A.1 Variables to bound with canaries

Which variables should be bounded with canaries? Following a high security policy, we should bound 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 is to setup boundary canaries after all *potentially vulnerable variables*. Such variables are the ones *for which a pointer is taken and used in the current function or other function called upon it.* Note that with this definition all local arrays are frequently vulnerable (unless they are not used or used only with constant indexes) but a pointer is not vulnerable until getting its address (see Figure 2). This policy is more efficiency then the previous one and should tackle most stack overflow problems (since they usually derive from deficient uses of legitimate stack addresses). Consequently, this was the policy we chose for installing our boundary canaries.

```
f ()
{
  int A, B;
  int * C;
  int D[10], E[10];
  ...
  C = &A;            // A becomes vulnerable
  f ( &B, C, D ); // B and D become vulnerable,
                     // but not C
  g ( &C );          // C becomes vulnerable
  E[A] = 3;          // E becomes vulnerable ...
}
```

Figure 2 - C function with 5 local variables, all of them potentially vulnerable to buffer overflows.

Additionally, all vulnerable variables are packed together to improve the detection mechanism. In fact, those variables and the canaries between them and at the end of the pack form a sort of "mined area", where most overflows, caused by dangling pointers and using either positive or negative offsets, are highly unlikely to occur without being noticed. The pack can either be placed close to the saved frame pointer or as far as possible; in principle, such decision should have no impact on the security provided by StackFences.

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

A.2 Looking for and checking canaries

How can we look for all or part of the canaries and how can we check the correctness of their value? One possibility is to generate and use per-function checking code, knowing the exact location of canaries and their values in the current stack frame. However, this approach would complicate a simple task of checking all existing canaries. Another possibility is to define a more function-independent way of locating and checking canaries.

We followed the last approach because it appeared to be more flexible and simple to implement and test. Consequently, the set of all boundary canaries form a linked list, as shown in Fig. 3. 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 neighboring variables to be tampered.

The full list of canaries, or particular sublists, 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

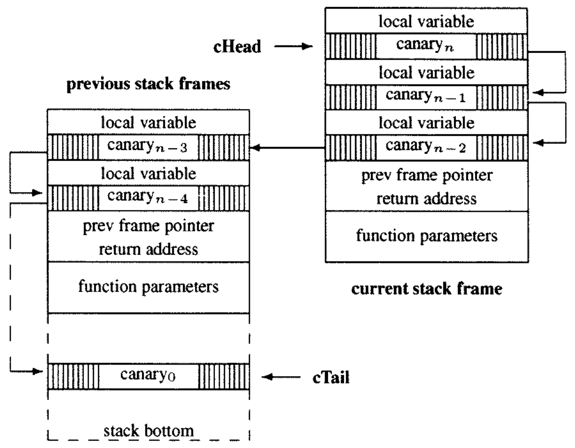


Figure 3 - Example of the list of canaries, starting in the current stack frame (canary$_n$) and until the first one inserted at the beginning of the process execution (canary$_0$). Variables `cHead` and `cTail` point the head and tail of the list. Shaded boxes represent canaries XORed with the process' XOR canary.

address that we want to reach. Any violation of these assertions is an overflow evidence.

A.3 Triggering canary checking

When should the application check the boundary canaries for looking for stack overflows? We think that 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 behavior. In other words, the coherence of stack variables should be checked before each and every operation capable of reflecting a wrong behavior of the application to the outside world. 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, that will disappear because the stack frame will be released. Note that a local overflow may have affected the past execution flow of the function, values returned by the function, variables modified by the function that are not bounded by overflow canaries (v.g. heap variables), or the frame pointer or the return address of the current stack frame.

In the first case, we should check the full list of canaries, because we are looking for any stack overflow. The full list of canaries, from the head `cHead` to the tail `cTail`, can be checked at any time during the execution of the program for finding out overflows affecting any of the canaries. 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. Checking the full list of canaries is a straightforward iterative walk, starting in the canary pointed to by `cHead` and ending when the canary pointed to by `cTail` is reached. Since canaries along the list must have strict growing addresses and cannot be higher than `cTail`, any violation of these assertions is an overflow evidence.

In the second case, we should check only the list of canaries belonging to the local stack frame ($\texttt{canary}_n$ to $\texttt{canary}_{n-2}$ in Fig. 3) 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 in the previous stack frames ($\texttt{canary}_{n-3}$ in Fig. 3). The location of this canary must be supplied by the compiler.

B. Management of the canary list

The canaries in each stack frame are similar to local variables. The difference is that (i) they are invisible to application code, being only known and managed by the compiler-generated code and (ii) they have a mandatory initial value that depends on the address of the next canary and the value of the XOR canary. Therefore, managing local canaries follows many of the usual procedures used for ordinary local variables.

The stack space for canaries around local variables can be allocated when the offsets of local variables (from the frame pointer) are defined by the compiler. This is a compile-time action that does not incur any run-time overhead. The setup of the canaries should occur after the normal C prologue[4].

The linked list is increased after the prologue of a function and decreased when the function returns. Increasing the list consists of adding all canaries next to local variables to the head of the list, referred to by cHead, and setting a new value for cHead with the address of the new top-most stack canary (the one with lowest address, $\texttt{canary}_n$ in Fig. 3). Decreasing the list consists simply of setting the value of cHead using the address of the first canary of the previous stack frames (the one with highest address, $\texttt{canary}_{n-3}$ in Fig. 3).

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 and that canary should become 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 first canary (the one with lowest address) in the stack frame we are jumping into, or in some other stack frame below.

C. 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: (i) for development or testing purposes, the sooner the overflow is spotted the better, while (ii) for the execution of the program in production environments, preventing the application from "making damage" may be enough for most cases.

[4]In order to use a correct frame pointer, or stack pointer in compilations omitting stack frames, to setup their values.

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.

C.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 fail to detect some local overflows. Note that an overflow in a stack frame may compromise memory areas all over the program if local memory pointers where affected (case 1 of Fig. 1). Consequently, the reduction of the canary list, both within a normal function return or within a call to longjmp, always checks the consistency of all released canaries.

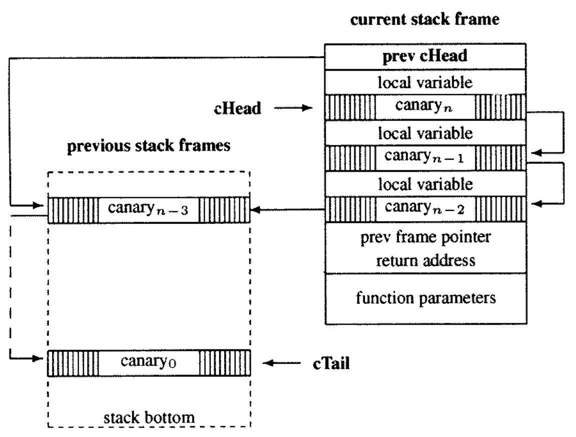
The extra code for checking local canaries and reducing the list of canaries was placed in a function (canReduce) that is called before the function's epilogue. All registers used to carry the return value of the ending function — EAX for 32-bit integer values or EAX+EDX for 64 bit integer values — are saved in the stack before calling canReduce and recovered afterward. canReduce gets, as a parameter, the address of the canary after the last one of the current stack frame (the address of $\texttt{canary}_{n-3}$ in Fig. 3).

For passing the correct value to canReduce we should not use the value stored in the last canary of the current stack frame because it may have been modified and, yet, be apparently valid. Being apparently valid means, in this context, that (i) it points to an address higher than its own and (ii) its value is not higher than the tail canary address (given by cTail). Thus, for a more accurate validation of canaries we store a clear copy of the previous head of the canary list at the top of each stack frame that adds canaries to the list (see Fig. 4). The function canReduce receives the previous head as parameter and follows the canaries from the current head until reaching the previous one; at the end the previous head will be stored in cHead. Overflow attacks cannot correctly corrupt both copies of the same value: one copy XORed with the XOR canary and another copy in clear, i.e., not XORed with the XOR canary ($\texttt{canary}_{n-2}$ and prev cHead, respectively, in Fig. 4).

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. Note that this approach does not require any modification of the jump context stored by setjmp and used by longjmp. It only requires an extension of the longjmp functionality.

C.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



```
...
#--- Check/remove local canaries
push   %eax
mov    prev cHead off(%esp), %eax
push   (%eax)
call canReduce
pop    %eax
#--- end of canary processing
leave
ret
```

Figure 4 - Epilogue of function (returning a 32-bit integer value) with extra code to check local canaries and remove them from the global list.

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 favor 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.

The rationale for the development policy is the following. We want to check the list of canaries often but we need to define exactly how often, when and why. By checking before calling a function we can do it quite often and still prevent the program from doing any I/O after a stack overflow. We are assuming, of course, that for doing I/O it is necessary to call a function (e.g. a system call library function) but this is not completely true. In fact, I/O can happen without actually calling any functions. For instance, we can use memory-mapped I/O objects, like files, and do I/O using pointers and read/write memory accesses. But we believe that most I/O, even for memory-mapped objects, is done using functions of utility libraries, thus enabling our development approach.

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.).

This approach should be faster than the previous one, since it implies fewer calls to `canWalk`. Again, any I/O using pointers and memory-mapped objects will not trigger the checking of all boundary canaries. Thus, it may be possible to perform I/O operations after a stack overflow without notice. Nevertheless, the overflow would eventually be detected afterward.

V. 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 don't need to be modified to use StackFences; they only need to be recompiled with the modified compiler and linked with some of the auxiliary modules. The C startup function `main` is transparently redefined for doing some setup actions in the auxiliary modules before actually starting the application.

For the C compiler we used the version 0.9.16 of TCC (Tiny C Compiler), a simple, complete and relatively easy to modify compiler developed by Fabrice Bellard [17]. We chose this compiler mainly because it allows a fast prototyping for getting a proof of concept. In the future we plan to modify `gcc`, a more popular but also more complex compiler, to support StackFences. For lack of space we will not describe here the modifications we did in TCC for implementing StackFences.

The first auxiliary module defines variables and functions that are common to all overflow checking facilities. Namely, it defines: the public variables `cXor`, `cHead` and `cTail`; the XOR canary setup function, the function `canReduce`; and a new `longjmp` function redefining the original one of the C library. For setting up a random value for the XOR canary this module uses the Linux special file `/dev/urandom`.

The second auxiliary module defines a new `main` function and all the checking and aborting functions for the development policy described in Section IV-C.2. The new `main` simply defines the initial boundary canary (canary_0 in Fig. 3), initiates the value of the variables in the previous module and calls the original `main` function of the application. None of the stack variables below the current stack frame are bounded with canaries; this means that program arguments and environment variables are still vulnerable. We decided this way because we believe that those variables can be protected more efficiently by other means, like memory protection mechanisms (see Section VIII).

The third auxiliary module defines a new `main` function and all the checking and aborting functions for the production policy described in Section IV-C.2. The new `main` first creates a child process for running the application and the initial process stays as the watchdog of the new one. The watchdog process will catch all system calls of the application process using the `ptrace` facility, typically used by debuggers and available in all Unix systems. The `main` function, when executing in the child process, is basically identical to the one of the second module. Again, stack variables below the current stack frame are not protected by boundary canaries.

For the watchdog action the third module includes a function for catching the system calls of the child process. This function triggers the 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, because it runs in a different process, canary values must be loaded from the target process with the system call ptrace and the request PEEK_DATA.

VI. EFFECTIVENESS EVALUATION

We tested the effectiveness of StackFences with the test suite developed by J. Wilander and M. Kambar, described in [9] and kindly provided by the authors. The results were the best possible: StackFences detected and halted *all* 12 attacks overflowing stack variables. We where also able to do so using either of the canary checking policies described in Section IV-C.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) [9, Tables 4 and 5]. 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 run-time 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.

There are other forms of attacks overflowing stack locations that are not performed by the test suite neither handled by StackFences. We are talking about overflows actually *using function parameters*, and not only considering them as targets. We believe that the general reason for researchers not considering such attacks is that overflows usually happen in arrays, typically for character strings, and C programmers usually pass arrays to functions by reference, not by value[5]. But we think that function parameters are as vulnerable as function variables and should be checked likewise, though that is not currently the case with StackFences.

VII. 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 could have devised microbenchmarks to study the influence of all these factors, but that would not help us to extrapolate the results in order to foresee the overheads introduced in real applications.

We decided to evaluate StackFences with both micro and macro benchmarks. The microbenchmarks 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.

All benchmarks ran in a Red Hat 8.0 Linux box, with the kernel 2.4.18-27.8.0, a Intel Pentium IV CPU at 2.4 GHz, 256 Mbytes RAM and 512 Kbytes cache.

A. Microbenchmarks

As microbenchmarks we used a set of functions with a variable number of variables, all of them vulnerable, and an expression per variable for triggering their vulnerability (see Figure 5). We measured the minimum CPU clock cycles taken by 100 consecutive calls of those functions.

```
void null (int * x) {}
```

```
void f0 () {}
```

```
void f1 ()
{
  int A1;

  // Turn A1 vulnerable
  null ( &A1 );
}
```

...

```
void f32 ()
{
  int A1, ..., A32;

  // Turn A1, ... and A32 vulnerable
  null ( &A1 ); ...; null ( &A32 );
}
```

Figure 5 - Example of functions used in microbenchmarks.

The microbenchmarks where compiled in 3 different ways — with gcc, tcc, and tcc with StackFences and the production policy. The results of the evaluation are presented in Table I: the first two columns show function names and their number of vulnerable local variables; the remaining columns show the minimum CPU clock cycles observed in 10000 consecutive runs of the benchmark. The values were measured using the Pentium's RDTSC instruction after a proper serialization with a CPUID instruction (as suggested by Intel [18]) and corrected by subtracting

[5]To pass an array to a function by value the array must be member of a structure an the structure must be passed to the function by value.

function	vulnerable variables	elapsed time (ms)			
		gcc	tcc		
			orig.	with StackFences	
f0()	0	3,520	3,528		(+0%)
f1()	1	4,524	4,532	6,732	(+49%)
f2()	2	5,724	5,736	8,960	(+56%)
f3()	3	6,940	7,004	10,556	(+51%)
f4()	4	8,136	8,136	12,576	(+55%)
f5()	5	9,340	9,332	14,388	(+54%)
f6()	6	10,536	10,824	16,512	(+53%)
f7()	7	12,148	11,932	21,212	(+78%)
f8()	8	12,928	12,928	26,656	(+106%)
f16()	16	22,756	22,536	47,988	(+113%)
f32()	32	41,724	41,768	92,956	(+123%)

Table I

RESULTS OF THE EXECUTION OF MICROBENCHMARKS FOR EVALUATING THE MAXIMUM OVERHEADS INTRODUCED BY STACKFENCES'S PROLOGUES AND EPILOGUES.

the minimum time taken by the measurement code. Therefore, they give a accurate notion of the maximum overheads imposed by StackFences in functions' prologues and epilogues. Note that these overheads are independent of StackFences's checking policies but the total overhead of StackFences per function used in the microbenchmark is not independent, because with the development policy we would call `canWalk` before actually calling the `dummy` function. To avoid such call we compiled the benchmarks only with production policy, as previously referred.

The microbenchmarks show that the maximum base overhead for protecting local variables is significant (approximately 50%) but the maximum cost per variable is relatively reduced. The base overhead is mostly due to the epilogue, namely to the `canReduce` function, because it implements a costly validation loop. StackFences's prologues are faster because they have no loops and no tests. The microbenchmarks also show that the overheads introduced by StackFences, when comparing with `gcc`, are not imputable to `tcc`. Thus, since StackFences's prologues and epilogues are almost compiler-independent[6], a similar overhead could be obtained with `gcc`.

B. Macrobenchmarks

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 checking policies — and executed with 3 different parameters — the sources of each of the 3 tools. For `ctags` and `tcc` we used a list of source files; for `bzip2` we used a `tar` file with the sources.

The results of the evaluation are presented in Table II: 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 optimization) and the normal `tcc` (that has no optimizations); 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 optimizations 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 II 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 [2], `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 development policy.

According to Table II, `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

[6]Actually StackFences's prologues/epilogues use the frame pointer, which is the common case and the only one supported by `tcc`. Thus, in compilations omitting the frame pointer, that are possible with `gcc`, they must be generated differently.

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

Table II

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.

StackGuard, which makes less validation actions! This paradox is probably explained by the fact that the performance figures presented in [2] 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 [2]. StackFences, on the contrary, was optimized to add canaries and for checking them locally with `canReduce` *only in functions with vulnerable variables*.

We believe that the overhead introduced by StackFences can be further reduced. For instance, different `canReduce` functions may be used for different lengths of local canary lists, allowing more effective loop unrolling optimizations. For reducing the overhead of the production policy we can keep the current watchdog model but use a thread, or Linux clone of the monitored process, to execute the function `canWalk`, or we can develop a kernel module or patch for tracing only the relevant system calls (see [19] for a detailed analysis and examples of this approach), instead of tracing all system calls from the watchdog process with `ptrace`.

VIII. CONCLUSIONS AND FUTURE WORK

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 this purpose StackFences extends two canary mechanisms to detect overflows: the canary mechanism introduced in StackGuard and the XOR canary mechanism introduced in StackGhost. StackFences uses canaries for monitoring all local variables susceptible to overflow, either affecting highly sensible values, like return addresses, or not.

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, thus more suitable for production environments.

For testing StackFences we developed a prototype for Linux systems using TCC (Tiny C Compiler). C modules compiled with TCC and StackFences are fully compatible with the standard C libraries and with modules compiled with other compilers or compilation options. The modifications introduced by StackFences affect only the code inside each function.

In terms of effectiveness, StackFences detected and halted *all* the 12 attacks of the test suite developed by Wilander and Kambar [9]. 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`. Namely, (i) `propolice` only tries to reduce the impact of overflows in character arrays, while StackFences detects all overflows in other kinds of vulnerable stack variables (ii) `propolice` may not detect overflows within consecutive character arrays, as StackFences does. Therefore, StackFences is much better suited for assessing the correctness of applications in run-time than `propolice`, though that is not properly demonstrated by the test suite.

The performance of StackFences is acceptable but depends a lot on the compiled application. Microbenchmarks showed that StackFences's prologues and epilogues have a significant maximum base overhead of about 50% in the elapsed time, but a small overhead per each vulnerable local variable. Macrobenchmarks with 3 tools showed a overhead in the elapsed time between 22% and 189%, when us-