SSPFA

Marco-Gisbert, Hector; Ripoll-Ripoll, Ismael

Published in:
International Journal of Information Security

DOI:
10.1007/s10207-018-00425-8

Published: 31/08/2019

Document Version
Publisher's PDF, also known as Version of record

International Journal of Information Security (2019) 18:519–532
https://doi.org/10.1007/s10207-018-00425-8

REGULAR CONTRIBUTION

SSPFA: effective stack smashing protection for Android OS

Héctor Marco-Gisbert[1] · **Ismael Ripoll-Ripoll**[2]

Published online: 22 January 2019

Abstract
In this paper, we detail why the stack smashing protector (SSP), one of the most effective techniques to mitigate stack buffer overflow attacks, fails to protect the Android operating system and thus causes a false sense of security that affects all Android devices. We detail weaknesses of existing SSP implementations, revealing that current SSP is not secure. We propose SSPFA, the first effective and practical SSP for Android devices. SSPFA provides security against stack buffer overflows without changing the underlying architecture. SSPFA has been implemented and tested on several real devices showing that it is not intrusive, and it is binary-compatible with Android applications. Extensive empirical validation has been carried out over the proposed solution.

Keywords Security · Buffer overflow · Stack smashing protector · Mobile devices · Android · Defenses

1 Introduction

The increase in the number of mobile devices relying on the same operating systems, Android OS and iOS, brings therefore, an increase in the exposition of operating systems against the discovery of new vulnerabilities, and thus, the possibility of using them against large-scale cyberattacks by exploiting such vulnerabilities [1]. In this sense, it is of paramount importance to keep revisiting and improving their security mechanisms to respond to the always evolving threats.

In this contribution, we focus our research efforts on Android OS due to its open-source license rather than a closed-source approach associated with iOS. Android OS, developed by Google, is based on the Linux kernel with some specific device drivers for mobile equipment and a 'C' library for embedded devices, namely the *Bionic* library used to allow developers to make use of the primitives exposed by the operating system. Android provides a security model for the execution of mobile applications where it is assumed that the mobile equipment will run a variety of untrusted or partially trusted applications. Then, Android tries to achieve isolation between applications by executing each application in a separate process with a different *User ID* per application and by making use of virtual memory addresses so that every application sees a different confinement memory area per application.

A weakness in the current security model architecture for the execution of mobile applications in Android OS will unveil vulnerabilities affecting all the versions of the operating system, all models of mobile equipment and all applications running in such operating systems and thus allowing millions of mobile devices to be potentially exploited, to be spied, to be used as potential attacking tool for cyberterrorism, just to name a few. This impact on the society has been in fact our main motivation, and the main contribution of this research work is exactly the identification and empirical demonstration of such significant weakness in the security model for the execution of mobile applications in Android OS together with the proposal of an enhanced memory protection architecture to protect mobile equipment against such weakness, thus protecting final users against cyberattacks exploiting such vulnerability.

The rest of the paper is organized as follows. Section 2 provides a detailed related work on memory protection techniques in Android OS. Section 3 describes Android memory architecture. Threats and vulnerabilities caused by the Android framework in relation to stack smashing protection (SSP) technique are described in Sect. 4. Section 5

✉ Héctor Marco-Gisbert
hector.marco@uws.ac.uk
http://hmarco.org

Ismael Ripoll-Ripoll
iripoll@disca.upv.es

[1] University of the West of Scotland, Paisley, UK

[2] Universitat Politècnica de València, Valencia, Spain

describes the proposed modification to SSP to overcome deficiencies identified, while the implementation of the new technique (SSPFA) is presented in Sect. 6. Section 7 provides an empirical evaluation of the proposed implementation. The paper finishes with a discussion on the general applicability of the proposed technique in Sect. 8 and some conclusions and future work 9.

2 Related work on memory protection architectures

Buffer overflow attacks such as heap overflows [2] and *stack smashing* have been one of the most dangerous threats to the security model of operating systems. Stack smashing attacks [3] consist of filling the stack buffer of a running program with data supplied from an untrusted user so that such user can corrupt the stack in such a way to inject executable code into the running program and thus take control of such process. This is one of the more reliable methods for attackers to gain unauthorized access to a computer.

Several techniques have been developed to mitigate the possibility of exploiting this kind of programming fault [3–6]. Stack smashing protection (SSP), address space layout randomization (ASLR) and No-eXecute (NX[1]) are widely used in most systems due to their low overheads, simplicity and effectiveness as effective ways to prevent such type of attacks. In fact, when these techniques are correctly implemented, they prevent or mitigate stack smashing attacks, execution of return-2-x [7] and return-oriented programming (ROP) [8–11] attack and code injection, respectively. Other techniques such as Flow Control Integrity [12], SmashClean [13] and Timing Channel Protection [14] have also been proven effective.

Unfortunately, it is not always possible to implement these techniques correctly. For example, ASLR [15–17] is only partially implemented, i.e., not all memory areas are randomized, including Android version 4.0 [18] and earlier, or are randomized only at system boot, as it happens with Mac OS and all versions of Android OS. With respect to SSP, one of the main problems (even in systems where it is correctly implemented) is *byte-by-byte* [19] attack, explained later. This research work is focused on the security analysis of SSP implemented in Android OS. Saito et al. [20] provide a comprehensive survey of prevention techniques against memory corruption.

Android applications are written in Java and executed either in the Dalvik virtual machine (Android version < 4.4) or in the Android Runtime (ART) (Android version ≥ 4.4). The Android execution framework is composed of a set of applications, most of which are written in Java, which provide all kinds of high-level services to applications. SSP is a low-level mitigation technique implemented at the ABI (Application Binary Interface) level. In contrast to an API (Application Program Interface), which defines structures and methods that can be used at the software level, an ABI defines the structures and methods used to access external, already compiled libraries/code at the level of machine code. Thus, it can only be applied to native code. Although Android applications are mainly written in Java, there are some parts of them that execute native code via Java Native Interface (JNI). In fact, many libraries are written in C/C++ and export their services via JNI to Java applications to allow this binding between Java and C. Also, some application sections inside of Java are written in C/C++ (and native code is generated) to overcome Java limitations. In Android, all these native functions are part of the so-called Native Development Kit (NDK) [21].

The first proposal of SSP was presented by Cowan et al. [22]. SSP technique [23] is a compiler extension which is implemented by: first, keeping out of the stack memory, a randomly generated value referred as *reference canary* value. Then, every time a new function protected by SSP is added to the stack memory, a guard section of memory, generally referred as *frame canary*, between the protected region of the stack and the buffers used to store the local values of a programming function. The value inserted in this guard is a copy of the reference canary. Then, every time the execution of a function is finalized, the value of the frame canary is checked against the value of the reference canary to determine whether the code has been exposed to a buffer overflow attack or not. If their values match, then no alternation happens and thus the return memory address stored in the stack can be used to continue execution flow of the program; otherwise, the program has been altered and thus SSP will stop immediately the execution of the program by raising the `SIGABORT` signal. The full explanation of the state of the art in the design of the canary guard is lately explained in Sect. 3.

Notice that the effectiveness of this SSP technique resides vastly in how efficiently the management of such reference canary value is performed in the security architecture. If the management related to the generation of such reference canary is not appropriate in a particular security architecture, a technique that is known to be effective can provide a dangerous false sense of security that can be easily exploited by attackers. In fact, this weakness can remain latent for a prolonged period, which in turn allows the attacker to prepare multiple assaults and tools that effectively bypass barriers that are generally considered as unbreakable (or properly settled). This is exactly the main motivation of this contribution where authors have demonstrated that the way Android manages today the reference canary value within its own security architecture makes this originally validated

[1] Also known as data execution prevention (DEP) , and Write xor eXecute (W ^X).

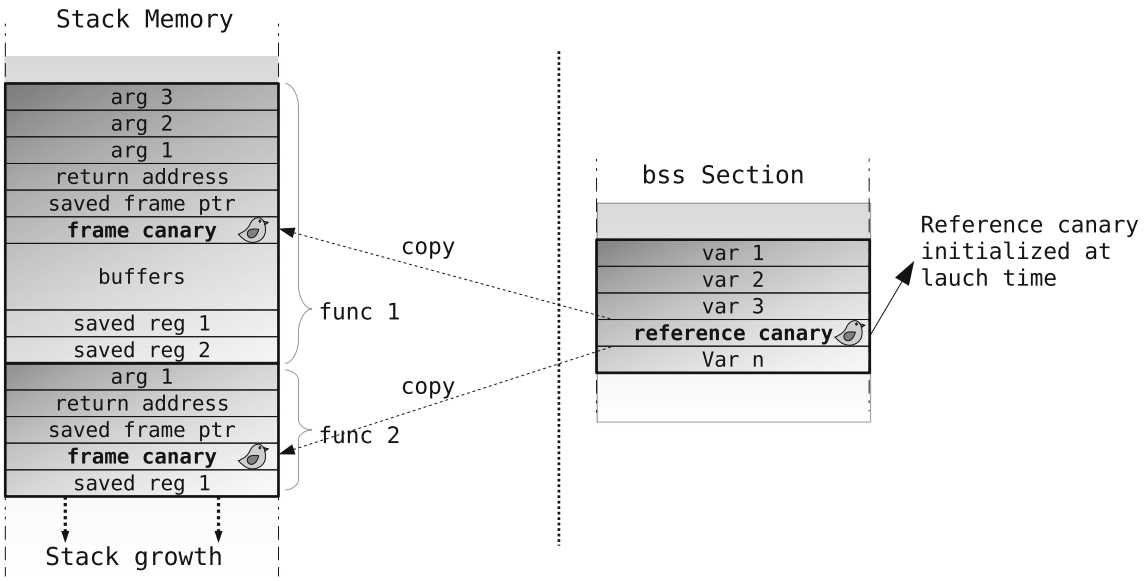

Fig. 1 Reference and frame canaries in the ARM architecture

and demonstrated technique weak and not suitable against buffer overflow attacks. When canaries were introduced by Cowan et al., they stated 'if the canary word is completely static, then it is very easy to guess' and they also stated 'these random words are then used as distinct random canary words, one per function in the object code.' SSP implemented in the Android security model uses static canaries ignoring the advice in Cowan et al. [22], and our proposal is rectifying this mistake. This is in fact the main finding of this publication.

In terms of solutions to the vulnerability of the security architecture of the Android OS against stack smashing attacks, B. Lee et al. [24] proposed to replace the *fork* model currently used in Android where all the Android applications are forked from a parent process by a *fork+exec* security model. This approach is very secure, and it is in fact used in highly secure products as OpenSSH [25]. They tried to apply this concept to Android. Unfortunately, as authors point, it has a prohibitive time overhead (more than 3.5 sec per launched application) and memory consumption (more than 13 MB per application). The temporal overhead is addressed using a pre-forked approach, which increases slightly further the memory consumption. Probably, this was the reason why Android developers discarded such *fork+exec* approach.

To the best of our knowledge, this contribution is the first one of its kind where it has demonstrated that the whole security architecture of Android OS is vulnerable against stack smashing attacks, which it is an attack supposed to be mitigated nowadays, such as other operating systems, including Linux OS which Android OS is based on, and thus affecting to all applications, all Android versions and in consequence all Android users. The identification of effective mitigation techniques to protect Android users has been the main motivation of our research work.

3 Overview of stack smashing protection architecture

Figure 1 sketches the layout of a stack with two stack frames (e.g., a function has called another function thus creating two stack frames). The reference canary is stored in the data segment of the memory of Android Bionic library, and it has been represented with a bird icon on the right side. Then, frame canary guard has been also painted with a bird icon available on each stack frame. With this memory layout, the scalar variables cannot be overwritten due to a buffer overflow because they are in lower addresses and buffers are typically overflown toward higher addresses. SSP only protects previously saved stack frame pointer, if the code is compiled with it, and return address.

The compiler emits extra code in the prologue and epilogue of each protected function for initializing the frame canary and checking this value against the reference canary, respectively. The value of the canary is chosen such that it prevents, when possible, the effective exploitation of a buffer overflow and detects the occurrence of an overflow.

Attending to these goals, four types of reference canary values have been proposed:

- Terminator value: The value is composed of different string terminators (CR, LF, NULL and − 1).
- Full random value: The value is randomly generated during the initialization of the process and stored in a global variable out of reach of the attackers; for exam-

ple, in x86_64 it is stored in the TCB[2] which resides in a separated memory space (segment `%fs`). The attacker needs to know the current value for building the exploit. As far as the value is kept secret, the attack will be prevented.
- Almost random value: The value is also randomly generated during the initialization and stored in a safe place, but the first byte is set to zero. This way, overflows caused by string handling functions, like `strcpy()`, are blocked because the copy operation terminates at the zero byte. Unlike the other types of canaries, this can not be bypassed knowing its value, if the failure is caused by a chain operation. This works in a similar manner as the ASCII-armored [26] technique against ret2lib attacks.
- Random XOR: are random canaries that when copied into the frame canary it is XORed using a control data register, instead of using a dedicated register, the stack or the base register is usually used. In this way, once the canary or the control data are modified, the frame canary value is wrong, which gives additional protection, but increases the overhead as it involves more operations.

Linux compilers (gcc and clang) use a full random word or a random word with a byte set to zero. Terminator value canaries can be trivially bypassed if the error is caused by an incorrect non-string memory copy. Therefore, this type of canaries is not used. Although XOR canary seems to be more effective because the value of the frame canary may be different on every function (depending on the control data used to XOR), the effective protection is close to the full random canaries [27]. XOR canaries are used in Visual Studio since version 2003 and hardened versions of Linux. Regarding the SSP implementation in the Android Bionic library, it uses a fully random canary (that is, all bytes of the canary are random values).

Initially, the canary frame guard was placed immediately after the return address since it was the target of most attacks, trying to alter such address using a buffer overflow attack to allow the redirection of the execution flow of the running program to another section of memory with the intention to execute malicious code.

New attack strategies have been developed (see Sect. 4) which have motivated some enhancements [28,29] over the original proposal. As of GCC v4.7, the latest version while writing this contribution, the stack smashing protector consists of the following:

- Both return address and saved previous stack frame pointer which are guarded by the frame canary;
- Local variables which are reordered so that buffers are located first[3] (higher addresses) and below them the scalar variables and the saved registers.

Since the value of canary is not a constant but a random value is chosen when the program starts, that value (the reference canary) has to be stored somewhere in the program memory or in a dedicated processor register if available. In the ARM architectures, the reference canary is stored in main memory by means of a global variable, called `__stack_chk_guard`, while in x86_32 and x86_64 it is stored in a different memory segment, `%gs:0x14` and `%fs:0x28`, respectively.

Regarding which functions (stack frames) are protected, there are currently several options:

Only those with a local buffer (`stack protector`): Only functions with local arrays are protected. Data structures containing buffers and pointers are not protected. Functions with local arrays are typically a small fraction of all the program; for example, in the Linux kernel [30], they represent the 2.81%.

All the functions (`stack-protector-all`): All the functions, regardless the type and the number of local variables, are protected. This option was added to overcome the problem of data structures (and other tricks) which was an important breach to the SSP. This strategy is the most secure, but it has a non-negligible overhead.

Most functions (`stack-protector-strong`): In 2012, Google engineers [31] designed and implemented a new option as a balance between security and performance. All the functions that are prone to be exploited by some kind of overflow abuse, not only to arrays, are protected. Among other conditions, SSP is applied on functions that meet any of the following conditions: Any of its local variables address is taken as rhs[4] of an assignment or function argument, or it has a struct/union containing an array, or has register local variables. According to Ingo Molnar [30], 20.5% of the Linux kernel functions are protected.

Every block of code with a buffer: Recently, in 2013, IBM [32] patented a method of splitting the code of a function into code which contains string manipulation operations (which is supposed to be prone to buffer overflows) and code without that behavior. The stack protector guard is used in the region with the string operation, which is a clever way to reduce overhead on the stack protector technique as well as to narrow the exposure

[2] TCB: Thread Control Block.

[3] Without loss of generality, we will assume that the stack grows from higher addresses to lower ones.

[4] rhs: right- hand side.

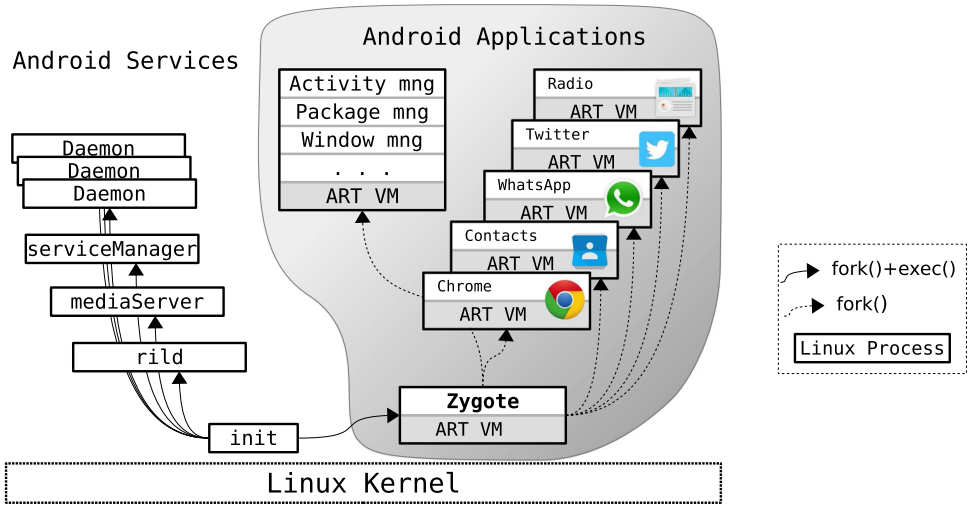

Fig. 2 Android execution architecture

window. To the best of our knowledge, this option is not implemented in any software.

Listing 1 shows the global variable used in Android under ARM architecture to store the reference canary value. The code is at the file `libc/bionic/ssp.cpp` of the Bionic library.

```
...   ...
035   #include "logd.h"
036
037   void *__stack_chk_guard =
    0;
...   ...
```

Listing 1 Android Management of the reference canary in ARM architectures

It is worth to remark that by default Android is compiled with the "only those with a local buffer" GCC option to perform a selective protection of ART functions.

4 Analysis of the SSP in the Android architecture

Figure 2 shows an overview of the security model implemented in Android OS. Zygote is an important Android process present in both Dalvik and ART security models, used mainly to speed up application launch. It is initiated at boot time by the Linux kernel to implement the security architecture of the Android OS. It contains commonly used shared libraries [33], application frameworks and the ART runtime framework. The ART runtime framework provides a sandbox environment where all the Android applications can be executed. This runtime framework provides some classes and resources that will be used by applications. Once Zygote is initialized, it will wait for commands on a socket. When a new application is requested to be launched, Zygote forks itself creating a new process and loading the application code in the prewarmed-up environment. Since most resources are already loaded in Zygote, the application can immediately begin executing. This way, the operating system can have significantly lower load times for the new applications, which is critical in low-performance execution environments such as mobile phones. Then, thanks to the copy-on-write mechanism, most of the system resources are shared between all the running applications until they are modified. Therefore, all processes forked from Zygote (i.e., all Android Applications) use the same copy of system classes and libraries. So, the Android architecture is based on the fact that Zygote is the father of all Android applications. Therefore, Zygote memory is copied to all applications. Thus, since it keeps the reference canary value in a global variable, the same reference canary value will be shared by all Android user applications running in the system. This characteristic has the undesirable effect that any local Android application knows the canary value of any other application. This is, in consequence, a potential vulnerability affecting all Android systems.

This vulnerability could be exploited by several different attack vectors. It is not the intention of this publication to provide all the possible vectors that can exploit the discovered vulnerability. However, in order to validate our claims, two different types of attack vectors have been analyzed in detail as suitable techniques to leak the value of the reference canary in Android OS: direct observation and brute force attack.

Direct observation: the value of the canary is directly read from the stack of the target process, or another place where is was copied.

Brute force attack: Different canary values are tried until the correct one is found. The attack known as byte-by-byte is an especially effective kind of brute force.

4.1 SSP direct disclosure

Every application running on an Android phone knows the system's reference canary value, and so attackers can add a simple but useful 'Trojanized' application [34] (examples of such apps are lantern, notes takers or simple but appealing games) to 'Google Play,' which sends to the attackers the value of its own canary value, jointly with other useful information.

Information can be sent directly from within the Trojanized application to the command and control [35]. But there are more subtle ways to do it, for example, as a bug report which contains a stack dump along with other process information.

Note that attackers can introduce legal applications which do not cause any damage to the system or try to launch an attack on other applications but just obtain local secret information. These applications only access their own data and do not require highly suspicious phone permissions. Some people care about the permissions granted to an application, but in this case no Android permission other than Internet communication is required to release the system's reference canary value.

4.2 SSP brute force attacks

In order to build a brute force attack, four conditions must be met:

1. Attacker must guess the secret.
2. Attacker has to be able to decide whether it is a correct or an incorrect guess.
3. The guess can be repeated as many times as needed by the attacker.
4. The secret value must always be the same. That is, it must not change during the attack; otherwise, tried and failed values cannot be discarded.

Let us focus on the first two conditions. Let us assume that a pure Java-based Android application will not produce any bug in the memory management to allow an attacker to exploit a buffer overflow unless there is a bug in the Java VM. However, a lot of Android applications are not pure Java-based application and rather they have been implemented using Android NDK to quickly port Linux-based applications into Android. Then, only graphical interfaces are created using Java and thus compiled applications are linked using JNI. In this scenario, where native application is implemented in C and thus it requires to deal with memory management, a buffer overflow vulnerability could be easily exposed by buggy code. This buffer overflow can be remotely exploited if such application provides, for example, a TCP connection and it is implemented using the traditional approach where a new process is created to attend each of the requests received. There are lot of these applications in the Google Play such as SSH server application, remote desktop applications, mouse controllers, Wi-Fi-sharing applications, file-sharing applications, messaging applications, and many more.

Then, if any of these applications is vulnerable against buffer overflows, that vulnerability can be easily exploited by overriding the value of the stack canary frame value. Then, all the combinations can be tried so that the first condition is fulfilled. Then, to see whether the attempt is correct or not, the fact that an incorrect canary value will produce an abort() library call that will abort abruptly the current TCP connection can be used. However, if the stack canary frame value overwritten is correct, then the TCP connection will remain open and this will allow the attacker to know that this is the correct value of the system reference canary.

The third condition is typically given on how a traditional socket handle is implemented, usually forking a new process to attend a new client request. In this case, the main server does not directly attend a client requests, but instead it forks a child processes which are in charge of attending to clients. Each child inherits the socket from the client as well as most of the parent's state, which includes the reference canary value. Then, when the buffer overflow is attempted, only the fork() process is aborted but the parent still works and then a new request will produce the forking of a new process (with the same canary value).

Regarding condition four, considering that most users reboot a phone only when it is strictly necessary [36] (perhaps due to flight regulations, when installing major software releases, system hangs, runs out of battery, etc.), applications have the same reference canary value for very long periods of time, which increases exposure time.

Depending on the granularity of how the attacker can flood the buffer (word or byte overflow), there are two different flavors of the brute force attacks that can be applied to Android applications:

Full brute force: The frame canary word is overwritten on each trial. If the guessed word is not correct, the application detects the error and aborts. The guessed value is discarded, and then the attacker proceeds with another value until all possible values have been guessed. The number of trials to bypass the SSP in a 32-bit Android ARM system is $\frac{2^{32}}{2} = 2,147,483,648$ on average.

Byte-by-byte: This is a dangerous kind of brute force attack which consists of overwriting only one byte of the canary in each trial until the value of the target byte is found; the remaining bytes of the canary are obtained by following the same strategy. The system can be defeated with, at most, $\frac{4\times256}{2} = 512$ trials, which is a fairly low number.

4.3 Summary of weaknesses

The execution environment in Android applications jeopardizes the effectiveness of the standard SSP technique.

- Weak security control in the Google Play store makes it relatively easy to upload malicious applications [37], which in turn are installed by careless users. Therefore, contrary to desktop and server systems, local attacks on smart phones (specially on Android) represent a main attack vector.
- Current Android SSP implementation is completely useless against local attacks, because the canary value is not a secret to local applications.
- Zygote as well as other system applications has the same broken SSP implementation.
- Remote attacks that must bypass the SSP on a target application may first attack the weakest installed application to obtain the canary value, and then use the obtained value against the real target application. This attack strategy makes exploitable some applications that otherwise would not be vulnerable.
- There is a very long exposition time. Once an attacker obtains the canary value, they can use it if the system is not rebooted, which may be a fairly long period of time.
- The obtained canary value can even be used against applications installed after the canary value has been leaked.

The reader may notice that once the system reference canary value is known by the attacker, any buffer overflow bug available in any of the applications launched by Zygote, i.e., any running Android application, can be used to execute malicious code by altering the return address of the stack.

5 SSPFA: enhanced SSP for the Android architecture

The section proposes a novel memory protection architecture SSPFA for Android to provide enhanced security in an operating system.

SSPFA relies on the same SSP infrastructure as already implemented by Bionic and GCC, but the code of the Zygote application is modified to renew the reference canary on the child process right after the new process has been created (forked). It is important to note that the value of the reference canary of Zygote is left unchanged, and only the reference canary of the forked/cloned processes is modified.

In order to understand why this modification does not break the normal operation of the application and allow backward compatibility to all the Android releases, the following observations shall be considered:

- In Zygote, after a `fork()` operation, the child process executes a flow of code which ends with an explicit call to the `exit()` call; i.e., the child process does not return to the main flow of control but jumps to execute the specific child code, which in turn ends with a call to `exit()`.
- There is a single reference canary per process which is stored in a protected/separated area and initialized during process start-up.
- Integrity (i.e., checking the frame canary against the reference canary) is only done at the end of each function (or block of code), immediately before the returning instruction.
- Only the value of the frame canary of the current stack is checked against the reference canary.

The point in the Zygote application where the canary is changed defines a 'point of no return.' To be more precise, once the reference canary has been changed for the fork thread, any attempt to return from such fork thread to Zygote will not trigger a match between the stacked frame canary value and the reference canary value and thus the process will be aborted. This is a normal behavior of any of the Android applications since they do not need to return the execution pointer to Zygote once they are being executed and thus there is not any implication in terms of the flow control of the native Android applications.

Figure 3 shows graphically the division of the stack once the reference canary has been changed. As the reader can see, all the stack frames associated with the Zygote application keep the same reference canary value and then in the moment, Zygote creates the fork to allocate the application, the newly inserted function renewssp() is invoked causing a point of no return for those applications. It makes that every application has a different reference canary value and thus all the exploited that make use of this vulnerability can be mitigated.

Non-local jumping (i.e., setjmp/longjmp) is another form of control flow which disrupts the normal execution of a program. It is typically used as an exception mechanism to jump or restore back multiple levels of function calls to continue from an initial safe state. Since `longjmp()` code does not check the stack integrity of the current and the destination functions, it can be safely used after a reference canary change. However, care must be taken if the destination frame of the destination function (or the previous stacked frames) contains frame canaries with the old value. The way to solve

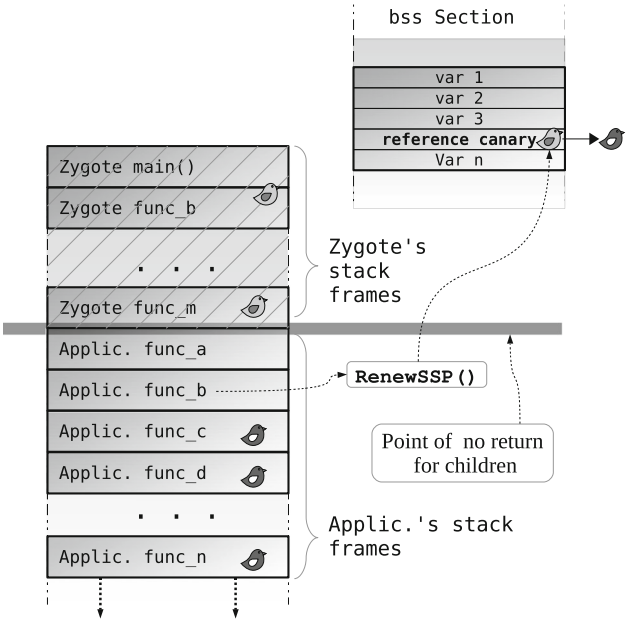

Fig. 3 Application active stack

this problem involves storing the value of the current reference canary when the `setjmp()` is called, along with the rest of environment information, and then restoring the reference canary to its original value when the `longjmp()` is later invoked. The value of the reference canary shall be considered a part of the execution context, and since it is not guaranteed to be constant throughout the execution of the process, it should be stored/restored when needed.

6 Implementation

This section described the specific implementation details performed in the Android platform to validate the proposed SSPFA memory protection architecture.

6.1 Application launch

There are three phases involved in launching a new process in Android OS: (1) process creation which creates the structures in memory to allocate the process, (2) application binding which associated the application compiled code with such process structures and 3) flow bypassing which passes the flow control to the application.

SSPFA implementation has been focus on the first phase (see Fig. 4). It is where the new application and the Zygote code depart from each other and where the reference canary value should be renewed. During this first phase, the Android `ActivityManager` sends to Zygote a request to create a new process via a connection socket. Then, Zygote forks a new process and instantiates the `ActivityThread` object, which starts specific application by calling the `onCreate()` function, which is the first call of the entire lifetime of the activity. We are only interested in the path of code executed from the `fork()` to the beginning of the application code.

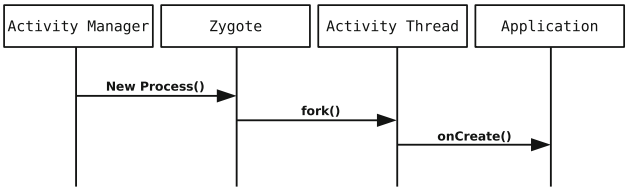

Fig. 4 Creation sequence of Android applications

These code sequences are executed once on the call sequence (by the parent process) and twice when returning (both the parent and the child process).

A detailed analysis of the code of these two sequences of calls shows that none of these functions is SSP protected, because they do not declare any local buffer. Therefore, it is safe to change the reference canary at any point during the execution of this sequence. In the case that any of them are protected by a canary, due to a future change in Android compilation options (e.g., compiler flags are changed to the more secure `stack-protector-all` or `stack-protector-strong`[5] flag), then they shall to be compiled with the `no-stack-protector` flag. A simple modification of the build scripts would fix this issue.

Once started, the application acts as a server that executes callbacks, and so parent functions are never returned from them. This behavior can be viewed as if there are two separate stacks—as shown in Fig. 3. The upper part of the stack contains stack frames with the old canaries, and the bottom part is the live stack of the application, which uses the new canary.

6.2 Application termination

During the normal execution of an Android application, the functions of the old stack are never returned from them. We analyzed how the applications terminate, to find out whether the application returns to the old functions or not.

Processes are activity containers, and their creation or destruction is controlled by the kernel. Android's execution model does not consider the termination of an application by calling `exit()` explicitly, and its full life cycle is beyond the scope of this work, but for our purposes it is enough to know that processes can be terminated in either of the following two modes:

[5] `stack-protector-strong`. is still a feature not available in the stable version of the GCC compiler, as of writing this paper.

(a) An application can call the method `Process.killProcess()` if the process is part of the application, or it can be killed by others if it has the '`android.permission.KILL_BACKGROUND_PROCESSES`' permission.
(b) Some versions of Android used a queue that keeps track of which applications have not been used. If the OS starts to run out of memory, it will kill an application (according to some metrics).

In both cases, the process ends by means of a signal. It does not return to any saved stack/environment, which meets SSPFA requirements.

6.3 Exception handling

Although Android is compiled using the C++ compiler, the code is mainly 'C'-compatible. Fortunately, the exception handling of C++ (i.e., try–catch blocks) is not supported by the Bionic library, which forces one to check errors and exceptions, by using explicit conditional constructions. Therefore, the stack is never unwound, due to a raised exception. This restriction causes the native code of Zygote (which is affected by the SSPFA) to be very procedural and sequential. Also, there are no calls for the setjmp/longjmp functions. On the other hand, the Java side of Zygote never checks the stack frame canary value against the reference so that Java exception handling is compatible with the SSPFA.

6.4 Modifications to Zygote

Once the impact of SSPFA on Android has been analyzed, the implementation is straightforward. We implemented the SSPFA for both Android 4.2 Jelly (with Dalvik VM security model) and Android 7.0 Nougat (with ART security model) as a proof of concept. We chose these versions of Android to cover both, Dalvik and ART.

The first step involves defining or giving access to a function which changes (re-randomises) the reference canary. We can reuse the already implemented function `__guard_setup()`, which initializes the reference canary with a random number from `/dev/urandom`. Rather than exporting this function, we preferred to export a dedicated function (called `renew_ssp()`), for clarity.

The code added to the mitigate the static reference canary required only one call to renew the reference canary (`renew_ssp()`) in the function `forkAndSpecializeCommon()` as shown in listing 2.

The function `forkAndSpecializeCommon()` is called from the functions `forkSystemServer()` and `forkAndSpecialize()`. These functions are used to launch new children for system services and general-purpose applications, respectively.

6.5 Implementation discussion

The modifications introduced by the SSPFA are not architecture dependent, so there are no restrictions on using our proposal on other hardwares supported by Android, such as MIPS or x86.

A key implementation issue is the source of random numbers. The function `__guard_setup()` reads four bytes from the `/dev/urandom` device. Urandom then produces an unlimited stream of random bytes, using a pseudo-random number generator, based on the internal entropy pool of Linux. Therefore, the four bytes of entropy consumed per application are not a problem at all. Also, we need to note that this is the default consumption rate on conventional systems, where applications are launched using the `fork()+exec()` pair.

This implementation exports the `renew_ssp()` symbol. This way, any native application compiled against the new library will be able to call that function at will. This function does not receive any parameter from the user, so it is impossible to reset (or set to a known value) the value of the reference canary. Therefore, security is not vulnerable when renewing the value of the canary at any moment during the execution of the process, as long as the already stacked stack canaries are not checked. A more general discussion about how and why to change the reference canary can be found in [38].

It is important to note that the old reference canary was used intensively by Zygote before forking. That value was pushed and popped from the stack multiple times while calling and returning from functions. Therefore, the stack of the child may still contain a copy of that value. These garbage values may reside in any location on the stack (downward or upward) and may be observed by a malicious application. Since Zygote does not change its own reference canary, the

```
490  static pid_t forkAndSpecializeCommon
     (...)
491  {
...      ...
553      dvmDumpLoaderStats("zygote");
554      pid = fork();
555
556      if (pid == 0) {
557          int err;
558          /* The child process */
559
560  #ifdef HAVE_ANDROID_OS
+++          renew_ssp();
561          extern int
     gMallocLeakZygoteChild;
562          gMallocLeakZygoteChild = 1;
...          ...
672      return pid;
673  }
```

Listing 2 `Zygote.cpp`

Table 1 Reference canaries with SSP and SSPFA

	App. Name	Process Name	SSP Canaries	SSPFA Canaries
Zygote 64-bit	Zygote64	zygote64	0xed82de32a74f858e	0x542a6326d3289ad1
	System	system_server	0xed82de32a74f858e	0xe5f7ef6577cdb8a5
	Phone App	com.android.phone	0xed82de32a74f858e	0x61c373b477f5c0a6
	Media App	com.android.media	0xed82de32a74f858e	0x75c8652655763a77
	MMS App	com.sec.imsservice	0xed82de32a74f858e	0x9c3d7367eb60477d
	Google Talk App	com.google.android.talk	0xed82de32a74f858e	0x7852ee0cd708407d
	Canlendar App	com.android.calendar	0xed82de32a74f858e	0xa3b77cd321d32f3d
	...	...	...	...
Zygote 32-bit	Zygote(32)	zygote	0x62650890	0x92342d21
	Chrome App	com.android.chrome	0x62650890	0x7ad32388
	Bluetooth App	com.android.Bluetooth	0x62650890	0xb7b21078
	...	...	...	...

Copied

Randomized

malicious application may be able to read the current canary of Zygote, albeit not the reference canary of the rest of the applications. The solution to this issue is to also change the value of the reference canary on the Zygote after a new process has been created. An analysis of how and where the canary of the Zygote shall be changed is beyond the scope of this paper, but we shall mention here that it can be done by following a similar approach to the one used to protect the applications.

Another interesting aspect to consider regarding the simplicity of the implementation is that it does not change the logic of Zygote, which greatly simplifies the maintainability on future versions.

7 Evaluation

The following aspects were evaluated: (1) the correctness of the modification, (2) overheads, both spatial and temporal, (3) portability and (4) effectiveness.

The correctness of the implementation was evaluated by running the system and reading the values of the canaries for the Android applications in both the original system and the one modified with the SSPFA. An overhead is only created because of the cost of reading four random bytes during application launch, and there are zero overheads during the execution of the application.

The evaluation of SSPFA, i.e., its effectiveness, was analyzed analytically, by comparing the operation of the current implementation with the new SSPFA. A detailed evaluation of the stack guard technique is beyond the scope of this paper.

7.1 Verification of the implementation

To show the feasibility of our proposal, a modified version of Android 4.2 and Android 7.0 has been built which includes the SSPFA. The implementation has been tested by reading the values of the reference canaries on the original version of Android and then on the modified version. The value of the reference canary can be read directly from the memory of the process, through the `/proc/ < pid > /mem`.

In Android 4.2, the reference canary is a global variable in the Bionic library, named `__stack_chk_guard`. In our example, it is located at the offset `0x4b228`. In Android 7.0, (in fact, from KitKat 4.4.4_r1 onward), the initialization of the canary has been moved into the Bionic core constructors.

The results relating to executing the inspector program in Android 7.0 are listed in Table 1. As expected, all of Zygote's children have the same reference value on a standard system but different values when using the SSPFA modification. It is worth to remark that in 64-bit ARM architectures there are two different zygote parent processes, one to fork 32-bit applications and another one to fork 64-bit applications and both are affected by the vulnerability as shown in Table 1. The main difference is the size of the frame canary which is 32 bits and 64 bits, respectively. The execution in Android 4.2 shows similar results but only with the 32-bit zygote architecture.

Table 1 also shows the canaries of native processes (those not launched by Zygote). In this case, the canaries are different because they are processes with a new binary image loaded by an `exec()` library call.

7.2 Memory footprint

The implementation of the SSPFA relies on the already existing infrastructure of the SSP and needs neither global nor local stack frame additional storage. Our implementation applies and exports the `renew_ssp()` function, which is just a proxy to `__guard_setup()` from *bionic* and adds a single call to this function in Zygote code. The function that has been modified in Zygote is not on the executable itself but in the shared library `libdvm.so`.

Table 2 Memory overhead (in bytes)

Library	SSP	SSPFA	Overhead
ARM			
Bionic	297,604	297,604	0
ART	710,280	710,280	0
MIPS			
Bionic	541,824	541,824	0
ART	1,342,944	1,342,948	4
x86			
Bionic	690,567	690,542	25
ART	1,393,859	1,393,884	25

The amount of code added is so small that the default optimization of function alignment to a 32-byte/64-byte boundary may hide the size of this additional code. Table 2 shows the different sizes of the libraries when compiled for ARM, MIPS and x86 architectures. As expected, it is negligible.

The size of the program is not increased at all in the ARM processor, due to alignment padding, which means that the SSPFA technique can be used on a mobile phone with zero memory overheads. And in the case of the x86, the global cost of the SSPFA is a total of 25 bytes. Note that this value is independent of the number of applications executed in the phone.

7.3 Temporal overhead

The temporal overhead is caused by the call to renew the canary on the child process after the fork operation, which is called only once per application. The rest of the execution of the application has zero overheads.

It is a continuously evolving sector in which devices are surpassed in a matter of months. It is therefore pointless to try to find a representative device for running performance tests, so we selected three phones according to their availability, namely Samsung Galaxy S4 mini and Huawei U8650 Sonic. Figure 5 summarizes the average cost of renewing the reference canary calling 100,000 times the function `renew_ssp()` on several devices against the cost of creating a new process using the `fork()` syscall.

Although the S4-mini is faster than the Huawei U8650, it took 38 μs versus the 26 μs of the Huawei U8650, because the kernel on that platform implements the SELinux facility, which adds an extra overhead to each system call, including the tree calls (`open()` , `read()`, `close()`) needed to read from `/dev/urandom`.

The cost of the SSPMD is approximately between 11 and 38 μs which is almost negligible compared with the start time of an Android application. It clearly shows that the overhead

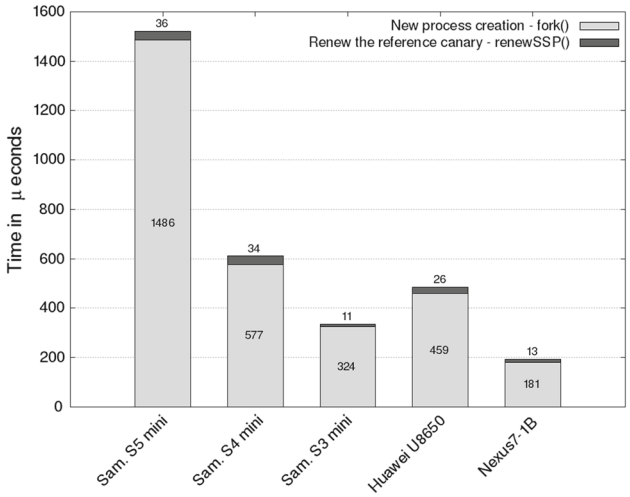

Fig. 5 Overhead of renewing the canary reference (μs)

of the usage of our proposed solution is really negligible and scales up significantly. Notice that the times shown are for calling 100,000 times the renew of the canary value and only one time is required to deploy our solution.

7.4 Portability

Although the implementation of the SSP is highly processor and compiler dependent, the SSPFA is not. Fortunately, neither the compiler nor the supporting library functions have to be modified, as all code modifications have been done in 'C' and in the generic part of the libraries. No platform-specific code has been added. Therefore, SSPFA is fully available to current platforms (ARM, MIPS and x86) and will be automatically available on new porting. Obviously, this transparency in the implementation greatly simplifies the maintainability on new releases for the same platform. SSPFA does not break any assumption or impose complex requirements or limitations on the Android architecture.

The only limitation is that once the reference canary has been changed, it is not allowed to return from previous stacked functions, if those functions check the canary. It should be considered that it is extremely rare to return back to the parent code after a `fork()` operation, and as far as the authors know, child processes execute another flow of code which always ends with a call to `exit()`. In order to validate this claim, we conducted an experiment which involved modifying the implementation of the Glibc `fork()`, to always renew (on the child) the canary. A complete Ubuntu 13.10 distribution, using this library, was used seamlessly.

Therefore, we can consider that the restriction required by SSPFA— that child processes must not return to parent functions—is not a limiting or an unacceptable requirement, because it is the normal default behavior of all analyzed appli-

Table 3 Android SSP versus SSPFA summary

Threat/issue	SSP	SSPFA
SSP full brute force attack:	Yes	No
SSP byte-by-byte attack:	Yes	No
ASLR brute force (on stack):	Yes	No
Direct disclosure bypasses:	All applications	Only the affected application
Local attacks are:	Trivial	Same as remote
Canary exposed until next:	Reboot	Application relaunch

cations. Table 3 summarizes in a few words the achievements of the SSPFA with respect to the current implementation.

8 Discussion

It can be argued that Android applications are not native applications but are instead byte code interpreted. Unfortunately, Android applications use native code through JNI. Note that the SSP technique is only applicable to native code; in fact, many libraries are written in C/C++ and export their services via JNI to Java applications. Some application parts are written in C/C++, to overcome Java limitations, for example to access system services that are not available otherwise (for instance, to interact with POSIX pseudo-terminals), to speed up critical parts or to reuse existing C/C++ code. SSP protection takes place in all native code used by Android applications.

Another aspect to consider is the applicability of the SSPFA. Although it may seem that the code where SSPFA can be used must meet very specific and somewhat odd conditions, a deep analysis of the code involved in how the `fork()` syscall is typically used reveals that most real applications meet those conditions by default. That is, the SSPFA can be used with minor modifications to Zygote, because it has been coded following standard programming patterns.

A solution to use SSPFA in every code, regardless the way the stack is used by children processes, was proposed by Petsios et al. [39]. The technique, called DynaGuard, performs a per-thread runtime bookkeeping of all the canaries that are pushed in the stack during execution of each thread. The buffer to store the frame canaries addresses is located in the heap, out of reach of the attacker. Unfortunately, the temporal overhead can be as high 3.2x, as pointed out by the authors. This high overhead is only acceptable when the security of the application is a must, and so, it is not practical for mobile applications.

In [38], the authors carried out an experiment to test the impact of the more general RAF-SSP technique on a full desktop system. The `fork()` function of the Glibc library was replaced by a custom `fork()` which always renews the reference canary on all children processes. A complete Linux distribution, using the modified library, ran smoothly. We cannot conclude that the RAF-SSP technique could be applied as a simple drop-in replacement for the fork, but it is very likely that a simple inspection of the code—and in some cases, a small modification—would be sufficient for its use.

Regarding the GNU GCC suite, Google engineers implemented the `stack-protector-strong`, which represents a balance between performance and coverage.

The SSPFA technique can be considered as another defensive measure which can be included in software, in an analogous way to other measures such as drop privileges, assertions and data canonization.

9 Conclusions

In this paper, we have detailed why the stack smashing protector (SSP), the most widely and effective technique used to mitigate attacks, fails on Android, the operating system used widely around the world, causing a false sense of security affecting millions of devices.

We detailed all weakness of current SSP in real attacks revealing that current SSP is not secure. We proposed SSPFA, the first effective and practical SSP for IoT Devices. SSPFA can provide security against stack buffer overflows without changing the underlying architecture.

We showed that the SSPFA prevents attacks against stack smashing protector. These core protection techniques are used to prevent modern attacks like BROP and Offset2lib.

We have implemented and tested the SSPFA on several devices, showing that it is not intrusive and that it is binary-compatible with all current and future Android applications. Applications do not need to be upgraded to benefit from the SSPFA protection. Contrarily to previous solutions that are not in use because they break the shared memory base concept used by zygote, SSPFA introduces zero temporal and spatial overhead which matches the requirements needed in the kernel space.

Funding This work was partially funded by Universitat Politècnica de València (Grant No. 20160251-ASLR-NG).

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Buchanan, W.J., Chiale, S., Macfarlane, R.: A methodology for the security evaluation within third-party android marketplaces. Digit. Investig. **23**(Supplement C), 88–98 (2017). https://doi.org/10.1016/j.diin.2017.10.002
2. Tian, D., Jia, X., Chen, J., Hu, C., Xue, J.: A practical online approach to protecting kernel heap buffers in kernel modules. China Commun. **1**, 143–152 (2016)
3. One, A.: Smashing the stack for fun and profit. Phrack, **7**(49) (1996)
4. Younan, Y., Pozza, D., Piessens, F., Joosen, W.: Extended protection against stack smashing attacks without performance loss. In: In Proceedings of ACSAC (2006)
5. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow Integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, Series CCS '05, pp. 340–353. ACM, New York (2005). https://doi.org/10.1145/1102120.1102165
6. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, Series CCS '12, pp. 157–168. ACM, New York (2012). https://doi.org/10.1145/2382196.2382216
7. Roglia, G.F., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: Proceedings of the 2009 Annual Computer Security Applications Conference, Series ACSAC '09, pp. 60–69. IEEE Computer Society, Washington (2009). https://doi.org/10.1109/ACSAC.2009.16
8. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1), 2:1–2:34 (2012). https://doi.org/10.1145/2133375.2133377
9. Pappas, V., Polychronakis, M., Keromytis, A.: Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 601–615 (2012)
10. S. R. to Thwart Return Oriented Programming in Embedded Systems, Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems, *IEEE Embedded Systems Letters*, vol. (first on-line), pp. 1–1 (2018)
11. Moula, V., Niksefat, S.: ROPK++: an enhanced ROP attack detection framework for Linux operating system. In: International Conference on Cyber Security And Protection Of Digital Services (Cyber Security). IEEE (2017)
12. Das, S., Zhang, W., Liu, Y.: A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. IEEE Trans. Very Large Scale Integr. VLSI Syst. **25**, 3193–3207 (2016)
13. Alam, M., Roy, D.B., Bhattacharya, S., Govindan, V., Chakraborty, R.S., Mukhopadhyay, D.: SmashClean: a hardware level mitigation to stack smashing attacks in OpenRISC. In: ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pp. 1–4. IEEE (2016)
14. Kananizadeh, S., Kononenko, K.: Development of dynamic protection against timing channels. Int. J. Inf. Secur. **16**, 641–651 (2017)
15. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a board range of memory error exploits. In: Proceedings of the 12th Conference on USENIX Security Symposium—volume 12, Series SSYM'03, p. 8. USENIX Association, Berkeley (2003). http://dl.acm.org/citation.cfm?id=1251353.1251361. Accessed 18 Jan 2019
16. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.-R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 574–588. IEEE (2013)
17. Kumar, K.S., Kisore, N.R.: Protection against buffer overflow attacks through runtime memory layout randomization. In: International Conference on Information Technology (ICIT). IEEE (2014)
18. Oberheide, J.: A look at ASLR in Android ice cream sandwich 4.0 (2012). https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0. Accessed 18 Jan 2019
19. Zabrocki, A.P.: Scraps of notes on remote stack overflow exploitation (2010). http://www.phrack.org/issues.html?issue=67&id=13#article. Accessed 18 Jan 2019
20. Saito, T., Watanabe, R., Kondo, S., Sugawara, S., Yokoyama, M.: A survey of prevention/mitigation against memory corruption attacks. In: 19th International Conference on Network-Based Information Systems (NBiS). IEEE (2016)
21. Meike, G.B.: Inside the Android OS: Building, Customizing, Managing and Operating Android System Services, illustrated ed., P. Education, Ed. Pearson Education, vol. 1 (2018). https://www.amazon.com/Inside-Android-OS-Customizing-Operating/dp/0134096347?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0134096347. Accessed 18 Jan 2019
22. Cowan, C., Pu, C., Maier, D., Hintongif, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium, pp. 63–78 (1998)
23. 'xorl': Linux GLibC stack canary values (2010). http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/. Accessed 18 Jan 2019
24. Lee, B., Lu, L., Wang, T., Kim, T., Lee, W.: From zygote to morula: fortifying weakened ASLR on Android. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, Series SP '14, pp. 424–439. IEEE Computer Society, Washington (2014). https://doi.org/10.1109/SP.2014.34
25. Miller, D.: Security measures in OpenSSH (2007). http://www.openbsd.org/papers/openssh-measures-asiabsdcon2007-slides.pdf. Accessed 18 Jan 2019
26. Molnar, I.: Exec shield, new Linux security feature (2003). https://lwn.net/Articles/31032/. Accessed 18 Jan 2019
27. Wagle, P., Cowan, C.: StackGuard: simple stack smash protection for GCC. In: Proceedings of the GCC Developers Summit, pp. 243–256 (2003)
28. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks (ProPolice) (2003). http://www.trl.ibm.com/projects/security/ssp/. Accessed 18 Jan 2019
29. Erb, C., Collins, M., Greathouse, J. L.: Dynamic buffer overflow detection for GPGPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 61–73 IEEE (2017)

30. Molnar, I.: Stackprotector updates for v3.14 (2014). https://lwn.net/Articles/584278/
31. Shen, H.: Add a new option "-fstack-protector-strong" (2012). http://gcc.gnu.org/ml/gcc-patches/2012-06/msg00974.html. Accessed 18 Jan 2019
32. Guan, X., Ji, J., Jiang, J., Zhang, S.: Stack overflow protection device, method, and related compiler and computing device, August 22 2013, uS Patent App. 13/772,858. https://www.google.com/patents/US20130219373. Accessed 18 Jan 2019
33. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in Android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Series. CCS '16, pp. 356–367. ACM, New York (2016)
34. Greenberg, A.: SC magazine: trojanized Android apps steal authentication tokens, put accounts at risk (2014). www.scmagazine.com/trojanized-android-apps-steal-authentication-tokens-put-accounts-at-risk/article/342208/
35. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Conference on Security, Series SEC'11, pp. 21–21. USENIX Association, Berkeley (2011) http://dl.acm.org/citation.cfm?id=2028067.2028088. Accessed 18 Jan 2019
36. Poll: How often do you reboot? (2014). http://www.androidcentral.com/poll-how-often-do-you-reboot. Accessed 18 Jan 2019
37. Wang, H., Li, H., Li, L., Guo, Y., Xu, G.: Why are android apps removed from Google play? A large-scale empirical study. In Proceedings of the 15th International Conference on Mining Software Repositories, Series MSR '18, pp. 231–242. ACM, New York (2018). http://doi.acm.org/10.1145/3196398.3196412
38. Marco-Gisbert, H., Ripoll, I.: Preventing brute force attacks against stack canary protection on networking servers. In: 12th International Symposium on Network Computing and Applications, pp. 243–250 (2013)
39. Petsios, T., Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: DynaGuard: armoring canary-based protections against brute-force attacks. In: Proceedings of the 31st Annual Computer Security Applications Conference, Series ACSAC 2015, pp. 351–360. ACM, New York (2015). http://doi.acm.org/10.1145/2818000.2818031