2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

To Detect Stack Buffer Overflow With Polymorphic Canaries

Zhilong Wang*, Xuhua Ding†, Chengbin Pang*, Jian Guo*, Jun Zhu* and Bing Mao*
*State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
{mg1633081,mg1733051,mf1733018}@smail.nju.edu.cn, clearscreen@163.com, maobing@nju.edu.cn
†School of Information Systems, Singapore Management University
xhding@smu.edu.sg

***Abstract*—Stack Smashing Protection (SSP) is a simple and highly efficient technique widely used in practice as the front line defense against stack buffer overflow attacks. Unfortunately, SSP is known to be vulnerable to the so-called byte-by-byte attack. Although several remedy schemes are proposed in the recent literature, their security is achieved at the price of practicality, because their complex logics ruin SSP's simplicity and high-efficiency. In this paper, we present an elegant solution named as Polymorphic SSP (P-SSP) that attains the same security without sacrificing SSP's strengths. We also propose three extensions of the basic scheme for better compatibility, stronger security, and local variable protection, respectively. We have implemented both a compiler plugin and a binary instrumentation tool for deploying P-SSP. Their respective runtime overheads are only 0.24% and 1.01%. We have also experimented with our extensions and compared their pros and cons with the basic scheme.**

***Keywords*—Stack buffer overflow; brute force attack; canary;**

I. Introduction

For decades buffer overflow remains as one of the main security threats plaguing the cyberspace, attributing to the prevalence of the software vulnerability it exploits and the low-level of complexity to launch it. Among all types of buffer overflow attacks, stack buffer overflow is arguably the most dangerous one, as it allows the adversary to tamper with the victim's control flow and computation results by modifying the return address and the local variables, respectively.

Stack Smashing Protection (SSP) is a well-known technique to detect stack overflow attacks and has been deployed on mainstream operating systems such as Windows and Linux for more than ten years [1]. It uses a random number called the *canary* loaded in the Thread Local Storage (TLS) during program loading and initialization. A function's prologue pushes the canary into the stack between the caller's saved return address and the callee's local variables; the function's epilogue compares the canary on the stack with the one in the TLS. A mismatch indicates that the canary is "killed" due to illegal writes out of the boundaries of the local variables on the stack.

SSP detects unintended stack buffer overflow with an overwhelming success probability. Nonetheless, its security against a determined adversary hinges on the secrecy of the canary. An exposed canary allows the adversary to craft his malicious input so that the resulting overflow does not violate the canary integrity. Since a canary is typically chosen as a 32-bit or 64-bit random string, its entropy is widely considered large enough to resist brute force attacks.

However, SSP has three noticeable drawbacks. As shown in blinded return-oriented-programming attacks [2], the byte-by-byte attack makes merely a few hundred trials to successfully recover the canary used by a network server. This efficient attack exploits the fact that the same canary is repeatedly used for all worker processes forked out by the network server. The adversary's strategy is to independently test correctness of each byte of the canary, so that its advantage is progressively accumulated. The other drawback is that it only detects those buffer overflows that tamper with the return address in order to manipulate the victim's control flow. It is incapable of detecting overflows that only affect local variables in the stack and leave the return address intact. Lastly, all stack frames of a thread share the same canary. Hence, if a vulnerability in one function exposes the canary, the adversary can overflow all vulnerable functions without being caught.

In this paper, we propose *Polymorphic Stack Smashing Protection* (P-SSP) to strengthen SSP from these three aspects. Our basic P-SSP scheme defeats the byte-by-byte attack by ensuring that the attacker's advantage of guessing the canary is not accumulated along with different trials. The core idea of P-SSP is to re-randomize the canaries for a new process/thread or for a new function call. The design of P-SSP ensures that the execution of the process/thread is not disrupted despite the fact that different versions of canaries co-reside in the stack. It also has backward compatibility, as P-SSP protected code can execute in the same control flow together with legacy binaries supporting SSP. As compared to existing schemes that also use refreshed stack canaries, our design is more elegant, because it does not have the hassle of tracking the function calls as used in those scheme.

We also make three extensions of P-SSP. The first extension provides better backward compatibility as it does not incur any change on the TLS or fork-like functions. The second extension expands the scope of protection to cover local variables. The third extension addresses the single-point-of-failure of SSP. It confines the damage of memory leakage in the sense that the knowledge of one function's canary does not lead to buffer overflow attacks on other functions.

P-SSP can be conveniently deployed in practice and incurs non-significant runtime overhead. We have modified LLVM

2158-3927/18/$31.00 ©2018 IEEE
DOI 10.1109/DSN.2018.00035

to support P-SSP. According to the benchmarks, programs compiled with the P-SSP option takes just 0.24% more CPU time than their native executions. We have also designed and implemented a tool to instrument existing binaries to support P-SSP. Executions with instrumented programs demonstrate a modest slowdown of 1.01% in average. Both implementations are publicly available as open-source projects.

ORGANIZATION. The rest of this paper is organized as follows: Section II discusses the background and related works. We present the overall design of P-SSP in Section III, and its three extensions in Section IV. We report the implementation of our systems in Section V. Section VI gives our evaluation of performance and effectiveness. We discuss several issues in Section VII and conclude the paper with Section VIII.

II. BACKGROUND

A. Stack Smashing Protection (SSP)

SSP is a default compilation option widely used nowadays as the front line defense against stack buffer overflow attacks. When compiling a function, the compiler produces the function prologue (as shown in Code 1) that pushes into the stack the so-called canary (a.k.a. *cookie*) which is a random number with the same length of a memory word. All functions in a thread share the same canary, as the function prologue copies the canary from the same location in the thread local storage.

```
1  push   %rbp
2  mov    %rsp,%rbp
3  sub    $0x10,%rsp
4  mov    %fs:0x28,%rax
5  mov    %rax,-0x8(%rbp)
6  ......
```

Code 1: Function Prologue of SSP

The canary is stored between the function return address and the local variables and is checked by the function epilogue (as shown in Code 2) when the function returns. Any buffer overflow of the local variables results in a different stack canary with an overwhelming probability. Hence, the epilogue code can detect the attack by spotting a different canary in the stack.

```
1  ......
2  mov    -0x8(%rbp),%rdx
3  xor    %fs:0x28,%rdx
4  je     Label
5  callq  <__stack_chk_fail@plt>
6  Label:
7  leaveq
8  retq
```

Code 2: Function Epilogue of SSP

On a 64-bit platform, the most naive adversary is expected to correctly guess the canary after making 2^{63} different trials. Although it is considered as secure against such a brute force attack, it is not secure against the byte-by-byte attack. We explain in detail how this attack works and then describe recent schemes proposed to cope with it.

B. Byte-by-byte Attack

The byte-by-byte attack is targeted at applications where a parent process keeps forking out child processes to undertake new jobs or serving new requests sent by external entities. With a large number of child processes, the application's performance is significantly boosted due to a high degree of execution concurrency.

The vulnerability of such applications is that the child process's TLS is cloned from the parent process when it is forked. Therefore, all child processes use the same TLS canary as the parent process. Moreover, the function prologue in SSP *copies* the TLS canary to its stack. As a result, all stack frames of all child processes share the same canary. Note that when one child process crashes, the parent process simply terminates it and forks out another child.

Intuitively, the byte-by-byte attack essentially treats the parent process as an "oracle" which tells the attacker whether its guess is correct or not. Specifically, it runs by gradually modifying the bytes in the stack canary, starting from the lowest address. It begins with overflowing only the lowest byte of the canary with other canary bytes unchanged. If the modification does not crash the process, the adversary is confirmed that the guess is correct. Then, it guesses the second lowest byte by overwriting two bytes of the canary with the correct lowest byte. In this fashion, it continues until all canary bytes are revealed. In average, the attacker needs to make $8 * 2^7 = 1024$ trials to break SSP in a 64-bit platform.

CAVEAT. Note that it is *not* an attractive approach to assign a new TLS canary to the child process. Due to the semantic of process forking, the child process starts to run with an inherited stack which contains frames belonging to the function calls made by its parent. A new TLS canary then does not match the canaries in those inherited stack frames. As a result, it disrupts the execution when the control returns to those functions.

C. Related Work

Several mechanisms have been proposed to protect the canary against brute force attacks. Gisbert et al. [3] implemented canary value randomization . Their technique, known as renew-after-fork stack smashing protection (RAF SSP), refresh the canary after the `fork()` function is invoked. Since it only updates the canary in the TLS area, this design suffers from execution failures when the child process returns to the frames inherited by the parent process. In order to ensure correctness, Petsios et al. [4] proposed and implemented DynaGuard to address the issue of inconsistent canaries in stack frames created prior to canary update. DynaGuard maintains a linked list of stack canaries during execution and updates the canaries in both the TLS and the old stack frame after a new child process is forked. DynaGuard can be deployed during compilation with a compiler plugin and at runtime using PIN [5] for dynamic binary instrumentation. Experiments with the SPEC CPU2006 benchmarks shows that compiler-based DynaGuard incurs 1.5% overhead while the instrumentation-based version incurs 156% overhead, as compared to the SSP protected executions.

Last year, Hawkins et al. [6] proposed and implemented Dynamic Canary Randomization technique (DCR). DCR and

TABLE I: Comparison of different brute force attack defence tools.

Defence Tools	BROP Prevention	Correctness	Runtime overhead (compiler-based)	Runtime overhead (instrumentation-based)
SSP	No	Yes	-	-
RAF SSP	Yes	No	negligible	negligible
DynaGuard	Yes	Yes	1.5%	156%
DCR	Yes	Yes	NA	>24%

DynaGuard differ in their means of maintaining the linked list of stack canary's locations at runtime. DynaGuard allocates a canary address buffer to store the linked list whereas DCR utilizes the existing canary reference storage space. To facilitate rewriting every canary value on the stack at runtime, in every canary of DCR, the offsets from address of itself to address of previous canary is embedded in it. These offsets are used by DCR to build a linked list of canaries on the stack during program execution. A pointer to the head of the list is stored in the TLS. DCR is implemented using the static binary instrumentation technique for re-randomizing every stack canaries on the old stack frame, with around a 24% overhead.

Table I compares the performances of the three schemes above and SSP. In short, existing schemes follow the approach of updating the TLS canary, which inevitably faces the challenge of maintaining canary consistency between the new TLS canary and the obsolete stack canaries. Our scheme presented in this paper takes a new approach which only updates the stack canary for new stacking frames without changing the TLS canary. Hence, our scheme does not have to deal with canary consistency and does not incur significant performance loss.

In a broad sense, our work is related to memory corruption and control flow integrity. We refer to readers the systemization of knowledge work by Szekeres et al. [7] for a comprehensive treatment. As noted in [7], although SSP does not provide the security assurance as strong as more sophisticated schemes like shadow stack [8] and CFI [9], it is still popularly used in practice due to its light overhead.

III. Main Design of Polymorphic Stack Smashing Protection

In this section, we begin with a description of the adversary model and our design goal. We then present the main scheme of Polymorphic SSP.

A. Adversary Model and Design Goal

The adversary in our model is the software attacker that feeds malicious inputs to the victim process in order to induce buffer overflows on the victim's stack. Depending on the service offered by the victim process, the adversary can be either remote or local. We do not assume secrecy of the victim's source code, the binary, or the virtual address space layout. Therefore, the adversary can adaptively choose the inputs and observe the outputs and behaviors of the victim process. Nonetheless, we have to assume that the adversary does not have the capability of direct memory read or write. With the capability of direct write, the adversary does not need to launch buffer overflow attacks, and the capability of direct read totally breaks the security premise of SSP.

Like SSP, we aim to provide a code-level stack buffer overflow detection mechanism which is secure against the adversary described above. The basic approach we take is to re-fresh the canaries in the stack so that information leaked in one attempt is invalid for other attack attempts.

Design Challenges. The main challenge of re-randomize the stack canary is to maintain the consistency with existing canaries in the stack frames, as all canaries are checked with the common one in the TLS. Prior work [6], [4] has demonstrated that it is time-consuming to keep tracking or to locate canaries in the stacks. Another challenge is to minimize the runtime overhead of canary checking. As noted by Szekeres et. al [7], the reason why SSP is popularly adopted in practice despite of its weak security assurance is its high efficiency and easiness to implement. Hence, it is imperative to preserve the performance virtues of SSP.

B. System Design

Without loss of generality, we consider 64-bit software and platform. The techniques we describe below can be easily customized for 32-bit platforms as well.

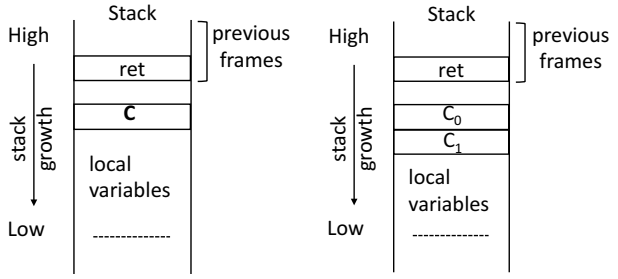
The Building Block: Canary Re-Randomization. Intuitively, our idea of re-randomizing the stack canary is to make the TLS canary randomly metamorphose to different forms as the stack canary, instead of cloning itself as in SSP. For easiness of description, we use $\mathbf{C}$ and C to denote the canaries stored in the TLS and in the stack, respectively. In SSP, we have $\mathbf{C} = C$, and both are 64-bit random binary strings. Our basic algorithm for canary re-randomization is described in Algorithm 1.

Algorithm 1 Re-Randomize($\mathbf{C}$):

INPUT: TLS canary $\mathbf{C}$;
OUTPUT: two binary strings of the same length as $\mathbf{C}$;
1: Generate a random binary string C_0 satisfying $||C_0|| = ||\mathbf{C}||$, where $||X||$ denote the binary size of X;
2: Compute $C_1 = C_0 \oplus \mathbf{C}$;
3: return C_0, C_1;

Clearly, the outputs (C_0, C_1) have the property that $\mathbf{C} = C_0 \oplus C_1$. Note that since C_0 is randomly generated, the exposure of C_0 does not leak any information about $\mathbf{C}$. Moreover, whenever the function is called, it produces a new pair of outputs which are bound to $\mathbf{C}$ like prior outputs, but are independent of them. In the following we present the Polymorphic SSP scheme (P-SSP) by using Algorithm 1 as the building block.

The Basic Scheme. We introduce the notion of *TLS shadow canary*, which is the outputs from re-randomizing the TLS canary $\mathbf{C}$. Whenever a child process is forked out, the system call handler copies the parent's TLS to the child process's. Then, it runs Algorithm 1 and sets (C_0, C_1) to the child process's TLS as the shadow canary. When a function is invoked in the child process, its prologue simply pushes C_0, C_1 into the stack between the return address and local variables. In other words, its stack canary C is in the form of $C_0||C_1$ where $||$ denotes binary string concatenation. When the function returns, its epilogue checks whether $\mathbf{C} = C_0 \oplus C_1$. If not, it signals the buffer overflow by jumping to the error handling code. Figure 1 compares the stack canary in SSP and P-SSP.



(a) Stack layout under SSP (b) Stack layout under P-SSP

Fig. 1: Comparison between stack layouts under SSP and P-SSP

CAVEAT. Although P-SSP updates the TLS, the TLS canary is *not* changed, which is different from all previous schemes. Hence, the semantics of forking is still preserved. When the control flow of the child process returns to the parent process's functions, the stack canary still matches the TLS canary, regardless whether P-SSP is used by the parent process or not.

C. Security of P-SSP

According to our adversary model, the attacker does not have the capability to directly read the stack canary. Therefore, we analyze the security based on the efforts for the attacker's (random) guesses.

1) Exhaustive Search: The most primitive attack is to exhaustively search the entire space of the canary. Since the stack canary's bit length is twice of the TLS canary's, the attacker randomly guesses the TLS canary. For each guess $\mathbf{C}'$, the attacker generates a random pair of C_0', C_1' satisfying $C_0' \oplus C_1' = \mathbf{C}'$, and overwrites the stack so that C_0' and C_1' replace the stack canary. P-SSP have the same security strength as SSP in terms of exhaustive search, because both schemes use the same amount of bits for the TLS canary. Hence, the adversary spends the same amount of efforts to successfully guess the TLS canary.

2) Byte-by-byte Attack: Recall that the byte-by-byte attack uses the strategy of guessing the individual bytes of the canary from the lower end. P-SSP resists the attack by using polymorphic canaries in the stack. Namely, a child process's stack canary is different from its parent, while their TLS canaries are the same. With P-SSP, every process fork uses a freshly generated stack canary which is independent from the byte(s) exposed to the attacker. Hence, the attacker's advantage is not accumulated after its random guesses. We summarize the security of P-SSP with Theorem 1.

Theorem 1. *Suppose that the adversary makes n rounds of attacks on n fork invocations. Let (C_0^i, C_1^i) denote the stack canary pair used in the i-th child process's stack frame. Then, the adversary does not gain any advantage in guessing the TLS canary, even if it observes $\{C_1^i | 1 \leq i \leq n\}$ of all child processes. Namely,*

$$\mathsf{Pr}(\mathbf{C}) = \mathsf{Pr}(\mathbf{C}|C_1^1, \cdots C_1^n) \tag{1}$$

Proof: (Sketch) The theorem can be proved by using an induction on n. Let t be the binary length of $\mathbf{C}$, C_0^i, and C_1^i for $i \in [1, n]$. Since $\mathbf{C}$ is a random binary string from the domain $\{0,1\}^t$, $\mathsf{Pr}(\mathbf{C}) = 1/2^t$. When $n = 1$, $\mathsf{Pr}(\mathbf{C}|C_1^1) = \mathsf{Pr}(\mathbf{C}, C_1^1)/\mathsf{Pr}(C_1^1)$. Since $C_1^1 = C_0^1 \oplus \mathbf{C}$, for every $\mathbf{C}$ randomly chosen from the domain of $\{0,1\}^t$, C_1^1 has the uniform probability of taking a value from the same domain, because C_1^0 is also randomly generated. Hence, $\mathsf{Pr}(C_1^1) = 1/2^t$, and $\mathsf{Pr}(\mathbf{C}, C_1^1) = 1/2^{2t}$. Hence, Equation 1 holds.

Suppose that the equation holds for $n = k$, it suffices to prove the theorem by showing that it also holds for $n = k+1$. Note that

$$\mathsf{Pr}(\mathbf{C}|C_1^1, \cdots C_1^k) \quad = \quad \frac{\mathsf{Pr}(\mathbf{C}, C_1^1, \cdots C_1^{k-1}, C_1^k)}{\mathsf{Pr}(C_1^1, \cdots C_1^{k-1}, C_1^k)}$$

According to P-SSP, whenever an erroneous canary is detected by a function's epilogue, that child process is killed and a new process is forked out with a fresh stack canary. Hence, C_1^k is independent of $C_1^1, \cdots, C_1^{k-1}$, we have $\mathsf{Pr}(C_1^1, \cdots C_1^{k-1}, C_1^k) = 1/2^{kt}$. With the same argument in the case of $n = 1$, we have $\mathsf{Pr}(\mathbf{C}, C_1^1, \cdots C_1^{k-1}, C_1^k) = 1/2^{(k+1)t}$, which shows that Equation 1 holds as well and therefore completes the proof. ■

D. Elegance of P-SSP

P-SSP is logically much simpler and tidier than existing scheme such as DynaGuard [4] and DCR [6]. The latter two's complex logics not only induce higher overhead, but also hinder their deployment due to the burden of maintaining compatibility. Namely, their requirement for a linked list of canaries in the process space ruins simplicity and transparency of canary based protection. Hence, exception handling, stack unwinding, across-modules function calls have to take those linked canary into consideration. It is therefore difficult to be applied for legacy binaries. In contrast, P-SSP does not cause the existing stack canaries to become invalidated by new stack canaries. Hence, the deployment of P-SSP results in no risk to program reliability.

IV. EXTENSIONS

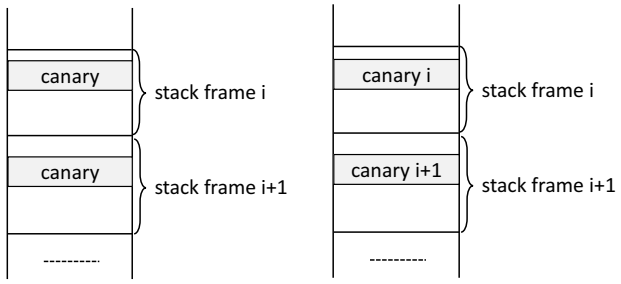
We propose three extensions of the main P-SSP scheme described in the previous section. The first extension avoids the need of updating the TLS structure during process forking; the second one expands the protection coverage from the return address to the non-control data, i.e., the local variables in the

stack; and the third extension addresses the single point of failure of using the same canary for all stack frames in a process.

A. P-SSP Without TLS Update

In this scheme (denoted by P-SSP-NT), we do not modify the semantic of forking or the structure of the TLS. As in SSP, the child process inherits the TLS canary from its parent without using any shadow canary. At runtime of the child process, each function prologue randomly and independently *splits* $\mathbf{C}$ in its TLS by using Algorithm 1, and pushes its own unique C_0, C_1 into the stack. The function epilogue remains the same as in P-SSP. Note that modern x86 processors have instruction support to generate random numbers. Hence, we do not need to inject a pseudo random number generation function to the prologues. We elaborate the details in Section V.

Comparison. The main difference between P-SSP-NT and P-SSP is the timing of re-randomization, i.e., upon function invocation v.s. upon process forking. The difference makes several implications. As shown in Figure 2, P-SSP-NT assigns different stack canaries to each stack frame while P-SSP uses the same stack canary for all stack frames in the process. The deployment of P-SSP requires modification of the fork implementation and the TLS layout, while P-SSP-NT does not. Hence, it is easier to deploy P-SSP-NT. In terms of runtime cost, P-SSP outperforms this extension, since its function prologue does not require random number generation which consumes more CPU cycles than memory copying.



(a) Stack frames use the same canary P-SSP

(b) Stack frames use different canaries in P-SSP-NT

Fig. 2: Comparison between stack layouts under P-SSP and P-SSP-NT. The shadowed regions denote the stack canaries.

B. Protection Over Local Variables

In comparison with tampering with the return address in the stack, the buffer overflow attack's modification on local variables is far more stealthy. We extend the P-SSP-NT to make a postmortem detection of such modification, in a similar fashion to return address protection. We name this extension as P-SSP-LV.

Consider that a function has n local variables denoted by $v_1, \cdots, v_n$. Without loss of generality, we suppose that the virtual addresses are sorted in ascending order with their indexes. Namely v_1 has the lowest virtual address and v_m has the highest. Among the n variables, let $\mathcal{V}$ be the set of critical variables that demand buffer overflow protection. The logic of the P-SSP-LV function prologue is shown in Algorithm 2.

Algorithm 2 function prologue for critical local variable protection

INPUT: TLS canary $\mathbf{C}$; $v_1, \cdots, v_n$; $\mathcal{V}$; $j = 1$;
OUTPUT: $m + 1$ canaries in the stack frame;

```
1: push the stack frame pointer into the stack;
2: /*to protect the return address and the stack frame pointer. */
3: generate a random number C_0;
4: push C_0;
5: /*to push local variables and protect those in V */
6: for i = n down to 1 do
7:    push v_i and set i=i-1;
8:    /* to generate and push canaries for critical variables */
9:    if v_i ∈ V then
10:      if j < |V| then
11:         generate a random number C_j and set j = j + 1;
12:      else
13:         /* the last canary has to be computed properly*/
14:         C_j = C ⊕ C_0 ⊕ ··· ⊕ C_{j-1}
15:      end if
16:      push C_j
17:   end if
18: end for
```

Similar to the protection over the return address, each critical variable is guarded by a distinct canary located in an adjacent memory word with a lower address. The algorithm of the prologue ensures that the XOR of all stack canaries in the function's stack frame is exactly the TLS canary $\mathbf{C}$. To avoid verbosity, we do not elaborate the algorithm of the function epilogue. Essentially, the epilogue is to check whether all stack canaries are collectively consistent with the TLS canary.

CAVEAT 1. We highlight the differences and advantages of P-SSP-LV over StackFences [10]. To protect stack variables, StackFences inserts same canaries for every *potentially vulnerable variables*. Hence, it can not protect program against brute force attacks. It is evident that the scheme is broken whenever a single canary is leaked to the attacker. However, for our scheme, the canaries are dynamically and independently generated. It is secure against brute force attacks. When one canary is exposed to the attacker due to one vulnerability, the attacker does not automatically obtain other canaries.

CAVEAT 2. We remark that the integrity of those critical local variables is *not* protected by the P-SSP extension, as those variables can still be updated by the program (or malware). In essence, the canaries in use protect the memory layout of those variables instead of their values, as only out-of-bound writes are detected.

C. Stack Canary Exposure Resilience

A common drawback of P-SSP and SSP is its single point of failure. If the stack canary in one stack frame is exposed (e.g., due to a memory leakage vulnerability in the corresponding function), the attacker can use it to successfully overflow all other stack frames. The deep-seated reason of the

ripple effect is that the exposure of one stack frame's canary leads to the exposure of the TLS canary, which means that the attacker can compose legitimate canaries for arbitrary stack frames.

We observe that a solution to deal with the problem must meet two requirements. Firstly, the adversary cannot derive the TLS canary from the stack canaries. Secondly, the stack canary has to be bound to its hosting stack frame so that it becomes invalid when being copied into a different stack frame. Based on this observation, the proposed extension is described in Algorithm 3 and named as P-SSP-OWF.

Algorithm 3 function prologue for exposure resilience

INPUT: the TLS canary $\mathbf{C}$, the return address ret;
OUTPUT: the stack canary C;

1: push the stack frame pointer into the stack;
2: get the nounce n;
3: push n;
4: compute $C = \mathcal{F}(ret||n, \mathbf{C})$ where $\mathcal{F}$ denote a one-way function.
5: push C;

Instead of randomly splitting $\mathbf{C}$ in P-SSP, the new algorithm evaluates a one-way function over $\mathbf{C}$, the return address, as well as a nounce n. Note that it is important to add the nounce n into the evaluation. Without the nounce being included, the stack frame will have a fixed canary that does not change with different executions because the one-way function itself is deterministic. Hence, it is subject to the byte-by-byte attack. From the cryptography perspective, the resulting stack canary C is a randomized message authentication code of the return address using $\mathbf{C}$ as the secret key. Hence, the knowledge of C in one stack frame does not leak the master secret $\mathbf{C}$. Neither is possible to forge a stack canary for another stack frame. The function epilogue runs in a similar fashion. It reads the nounce from the stack and re-evaluates $\mathcal{F}(ret||n, \mathbf{C})$ and checks whether the output is equal to the stack canary. A mismatch signals a buffer overflow attack and the control is transferred to the error handling code.

There are two methods to instantiate the one-way function $\mathcal{F}$, i.e., a hash function (e.g., SHA-1) and a block cipher (e.g., AES). Without hardware support, it is prohibitively expensive to evaluate $\mathcal{F}$ in every function's prologue and epilogue. Modern Intel processors already provide such a support. We report its performance overhead using Intel's AES-NI in Section VI.

V. Implementation of P-SSP and Its Extensions

To assess P-SSP's overhead and easiness of use, we implemented an P-SSP compiler plugin and a binary instrumentation tool which produces function prologues and epilogues supporting P-SSP-NT. In addition, we implement a shared library which is invoked when a new program is launched or a child process is forked. The shared library is to handle the updates on the TLS.

A. P-SSP Shared Library

Note that function prologues and epilogues do not update the TLS. Hence, runtime support is needed to load the fresh TLS shadow canary. There are two occasions for TLS creation: program startup and thread creation which is via the `fork` function in Linux. We have implemented a position independent shared library which exports three functions (i.e., `setup_p-ssp`, `fork`, and `pthread_create`) to override their counterparts in the standard GNU C library.

Function `setup_p-ssp` is defined with the `constructor` attribute. Therefore, it is invoked automatically before executing `main()` of a program. It basically initializes the TLS shadow canary according to Algorithm 1. Specifically, we use the addresses from **%fs:0x2a8** to **%fs:0x2b7** to store the TLS shadow canary C_0, C_1 and while addresses from **%fs:0x28** to **%fs:0x2f** hold the canary $\mathbf{C}$. Function `fork()` is called when a process create a child process. We wrap the glibc `fork()` function to refresh the TLS shadow canary after the child process's TLS is cloned from its parent's address space. Note that only the child process's TLS is updated. The last modified function is for thread spawning. Similar to `fork()`, the thread creation function is wrapped to refresh the TLS shadow canary.

The binary size of the P-SSP shared library is only around 16 KB, which is compiled from about 358 lines of source code. At runtime, it can be linked to existing binaries by using via the `LD_PRELOAD` mechanism.

B. P-SSP Compiler Plugin

We customize the Low Level Virtual Machine (LLVM) compiler [11] to support P-SSP. Specifically, we implement the P-SSP compiler plugin which is registered as one LLVM pass. We declare a `P-SSP-Pass` class that is a subclass of `FunctionPass` which is invoked on each function in the source code, perform instrumentation on the intermediate language (IR) [11]. `P-SSP-Pass` overloads a virtual `runOnFunction` method to perform its task. The `runOnFunction()` method decides whether to insert P-SSP canary according to the types and lengths of local variables. When there exists a local buffer in the stack, it generates both the function prologue and the function epilogue. Specifically, after traversing every basic block in the function, it creates the prologue at the beginning of function and create the epilogue right before each **ret** instruction.

We compile the pass into the dynamic library file `libP-SSP.so`, then register it in LLVM's *Pass Manager*. The function prologue and epilogue produced by *clang* are shown in Code 3 and 4, respectively. As described earlier, addresses from **%fs:0x2a8** to **%fs:0x2b7** hold the canary C_0, C_1 while addresses from **%fs:0x28** to **%fs:0x2f** hold the canary $\mathbf{C}$. In the function prologue, a 16-byte storage between local variables and the return address is allocated to hold a copy of C_0, C_1. The function epilogue verifies the C_0, C_1 in stack frame:

a. It loads C_0 and C_1 to register **rdx** and **rdi**, respectively;
b. It evaluates the exclusive OR of **rdi** and **rdx**, and the result is saved in **rdx**;
c. It compares the content in **rdx** with the TLS canary. A mismatch leads to a call to `__stack_chk_fail`.

```
1   push   %rbp
2   mov    %rsp,%rbp
3   sub    $0x10,%rsp
4   mov    %fs:0x2a8,%rax
5   mov    %rax,-0x8(%rbp)
6   mov    %fs:0x2b0,%rax
7   mov    %rax,-0x10(%rbp)
8   ......
```

Code 3: Function Prologue of Compiler based P-SSP

```
1   ......
2   mov    -0x8(%rbp),%rdx
3   mov    -0x10(%rbp),%rdi
4   xor    %rdi,%rdx
5   xor    %fs:0x28,%rdx
6   je     Label
7   callq  <__stack_chk_fail@plt>
8   Label:
9   leaveq
10  retq
```

Code 4: Function Epilogue of Compiler based P-SSP

C. Binary Instrumentation for P-SSP

We have also developed a tool to instrument legacy binaries to use P-SSP. Since SSP is one of the default compilation options (`-fstack-protector`) of GCC and LLVM, we assume that function prologues and epilogues in the target binary already contain SSP instructions. Comprising around 1100 lines of C++ code, our tool is essentially a binary rewriter that replaces the canary handling instructions emitted by SSP with P-SSP code in function prologues and epilogues.

Upgrading SSP to P-SSP in the binary level faces two challenges. The first challenge is that the instrumentation must preserve the stack layout which is the premise of stack related code. Any change on the stack layout is likely to disrupt the execution. For instance, instructions that access to local variable using a fixed offset from a "stack base pointer" (e.g., `ebp` and `rbp`) will make erroneous memory accesses if the stack layout is modified. While the first challenge is unique to P-SSP, the second one is common for binary instrumentation. It is to preserve the address layout of the target program. In other words, the instrumentation does not insert more bytes to the program than it deletes, so that the offsets of various sections and functions entrances are not affected.

To cope with the first challenge, we endeavor to strike a good balance between security, performance and compatibility. Note that the P-SSP scheme in Section III induces stack layout changes by adding one more canary to the stack as compared to the stack of SSP. Hence, we downgrade 64-bit canaries to 32-bit canaries. Namely, C_0, C_1 are 32-bit long and jointly form a memory word in the stack, so that the actual storage for the stack canary does not grow and the stack layout is therefore consistent with SSP stacks.

CAVEAT. We acknowledge the drop of canary entropy. Nonetheless, we argue that the drop does not give the adversary significant advantage in practice, especially for 64-bit platforms. After the failure of one round of attack, the canaries are refreshed. Therefore, the adversary constantly faces the challenge of breaking a 32-bit canary. For 32-bit platforms, the adversary is still expected to make more than ten thousand of trials to correctly guess the canary, which is still 64 times more than the byte-by-byte attack on SSP.

To cope with the second challenge, we consider function prologue and epilogue separately. The P-SSP function prologue code is exactly the same as the SSP prologue code, except the source of the TLS canary. The latter copies one 64-bit TLS canary from the address **%fs:0x28** while the former copies *two* 32-bit TLS shadow canary from the address **%fs:0x2a8**. Since both use the same **mov** instruction, our tool simply replaces the offset in use. Code 5 presents the P-SSP function prologue after instrumentation and Line 4 is the only instruction that is different from the SSP function prologue.

```
1   push   %rbp
2   mov    %rsp,%rbp
3   sub    $0x10,%rsp
4   mov    %fs:0x2a8,%rax
5   mov    %rax,-0x8(%rbp)
6   ......
```

Code 5: Function prologue of instrumentation based P-SSP

Handling the function epilogue is slightly more difficult because the P-SSP function epilogue has a more complex computation logic than the SSP epilogue, which implies that more instructions are needed. To avoid the inflation of the epilogue code size, we replace the comparison instruction with a call instruction to invoke the canary checking function whose parameters are passed via the **rdi** register. Therefore, the register **rdi** is saved before canary checking and is restored afterwards. Code 6 presents the epilogue function, which has the same length as the SSP epilogue.

```
1   mov    -0x8(%rbp),%rdx
2   push   %rdi
3   push   %rdx
4   pop    %rdi
5   callq  <__stack_chk_fail@plt>
6   pop    %rdi
7   je     Label
8   callq  <__stack_chk_fail@plt>
9   Label:
10  leaveq
11  retq
```

Code 6: Function epilogue of instrumentation based P-SSP

To minimize the modification on the binary, we combine the canary checking function with the existing `__stack_chk_fail` function. More specifically, we insert the canary checking code before those instructions handling the failure, as shown in Figure 3.

Since the epilogue code loads C_0, C_1 into the **rdi** register, the canary checking code is slightly from the compiler based P-

SSP epilogue, although the logics are the same. As illustrated by Figure 4, it firstly separates C_0 and C_1 in **rdi**. Then it computes the exclusive-OR of them and compares the result with the TLS canary. A mismatch causes the control flow to call the `__GI__fortify_fail` function which aborts the execution and reports an error; otherwise it sets the zero flag to true and returns to the caller.

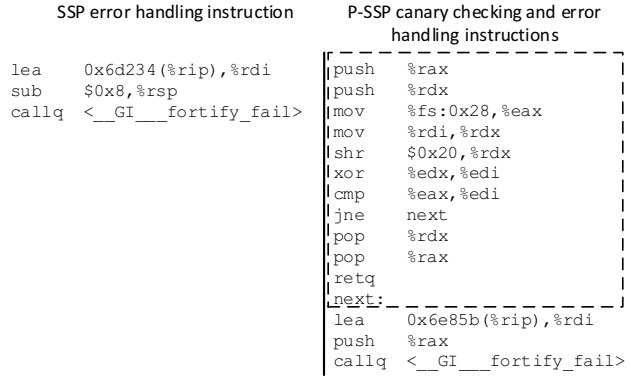


Fig. 3: Modification of __stack_chk_fail(). The instructions in the dashed box are inserted for P-SSP to check the stack canary.

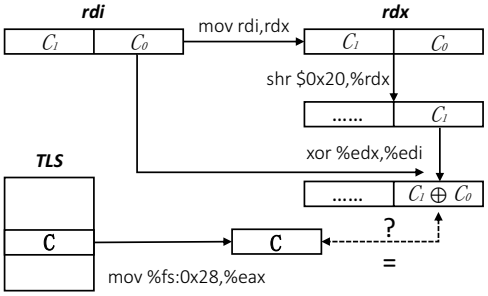


Fig. 4: Canary Check in `__stack_chk_fail` of Binary based DCSE.

Finally, we guarantee that the new `__stack_chk_fail` is also compatible with SSP. If the SSP detects a mismatching canary and invokes the `__stack_chk_fail` function, `rdi` is not equal to the TLS canary with an overwhelming probability. In such circumstances, the instrumented `__stack_chk_fail` eventually invokes the `__GI__fortify_fail` function.

D. Instrumentation On Statically Linked Code

Lastly, we describe how to handle statically linked glibc C functions that require instrumentation. Although it is not common, functions like `fork()`, `pthread_created` and `__stack_chk_fail()` may be statically embedded in the binary. These functions have to be modified to support P-SSP. We use Dyninst [12], a binary rewriting tool, to add a new code section to hold the customized `fork()` and `__stack_chk_fail()`. In a nutshell, Dyninst uses **jmp** instruction to hook the relevant function calls so that the control flow transfers to the customized functions at their entrances.

E. Implementations of Three Extensions

To assess the feasibility and performance of the three extensions, we have developed a prototype for each of them.

1) P-SSP Without TLS Update: In this extension, we write a compiler plugin to emit function prologues and epilogues. Note that the TLS structure remains the same as in SSP. It does not require a special shared library to overload `setup_p-ssp`, `fork`, and `pthread_create` in the glibc library. The changes on the binary are limited to function prologues and epilogues.

While The function epilogue of P-SSP-NT is the same as in P-SSP, its function prologue is different. The main difference is that we use the instruction `rdrand` to get a random number as the canary as shown in Line 3 of Code 7. Supported by both Intel and AMD processors, this instruction uses the entropy of the hardware resources to generate a random number.

```
1  push   %rbp
2  mov    %rsp,%rbp
3  rdrand %rax
4  mov    %rax,-0x18(%rbp)
5  mov    %fs:0x28,%rcx
6  xor    %rax,%rcx
7  mov    %rcx,-0x20(%rbp)
8  ......
```

Code 7: Function Prologue of Compiler based P-SSP-NT

2) P-SSP With Local Variable Protection: It is slightly more difficult to implement P-SSP-LV using a LLVM plugin. First of all, many compiler optimization in LLVM re-order the local variables in functions. To detect corruption between the local variables, we must place canaries at the address higher than each local buffer. Variable re-ordering therefore breaks the bond between variables and their canaries. One possible solution is to execute the P-SSP pass after all the optimization passes. Nonetheless, it may lose some benefits of compiler optimization. Another possible solution is that the P-SSP-LV pass encapsulate the local variable and its canary as a new struct type and replace every reference to the local variable with the reference to the struct's first member. Alternatively, the complier can append eight bytes space to the local variable and use the highest 8 bytes to hold the canary. However, it may affect functions like *sizeof()*.

Another issue is about the timing of canary checking. In SSP and P-SSP, the canary is checked by the function epilogue, namely at the moment of function return, because the canary is used to protect the return address. For local variable overflow detection, it could be too late to detect their overflow at function return. One design option is to inspect the canary when the variables are written by vulnerable functions. For instance, the plugin may add canary inspection code after executing functions like *strcpy()*, *read()*, *memset()*, *scanf()*, *strcat()*, *gets()* which may write data to a local variable. Since

these are widely used C functions, the compiler needs to determine whether a local variable is the target of writing.

Another design consideration is to select the variables for protection. One approach is to introduce a new data type and let the programer to specify the sensitive variables. Another approach is that the compiler discover sensitive local variables in the source code and insert canaries in adjacent addresses.

Due to these implementation challenge, we decide to leave the compiler based P-SSP-LV in the future work. In our present implementation, we manually identify sensitive variables and insert the corresponding P-SSP-LV function prologue and epilogue. Note that it is challenging to implement P-SSP for legacy binaries, because it totally changes the stack layout and references to local variables using the stack pointer become all invalid.

3) P-SSP With Exposure Resilience: We use Intel Advanced Encryption Standard New Instructions (AES-NI) to implement P-SSP-OWF. Because the minimum length of key and plaintext of AES-NI is 128-bits, We treat the canary in registers `r12` and `r13` as an AES key and use it to encrypt unpredictable data (Time Stamp Counter) and function level independent data (return address) in the function prologue and can check it in the epilogue.

To prevent the use of `r12` and `r13` for other purpose, we define key as global register variables in `r12` and `r13`. Besides, if a binary, compiled with P-SSP-OWF and therefore reserves `r12`,`r13`, invokes a shared library that uses `r12` or `r13`, the canary in register will not be covered. Because according to the calling conventions in Windows, Linux and Mac OS, `r12` and `r13` are callee-save registers. If the shared library use this register, it must save it firstly.

In the function prologue, as shown in Code 8, first of all, cpu cycles data is generated via Time Stamp Counter instruction `rdtsc` and loaded to the register `rax`. Then, we put both of 8 bytes of cpu cycle data and return address to `xmm15` as the data to be encrypted. Than we move the AES key in `r12` and `r13` to `xmm1` and call the `AES_ENCRYPT_128` to encrypt the data in `xmm15` and return ciphertext in `xmm15`. At Last, we place both of the cpu cycle data and the ciphertext onto stack.

```
1   push   %rbp
2   mov    %rsp,%rbp
3   rdtsc
4   shl    $0x20, %rdx
5   or     %rdx, %rax
6   movq   %rax,%xmm15
7   movhps 0x8(%rbp),%xmm15
8   movq   %r13,%xmm1
9   punpckhdq %r12,%xmm1
10  callq  <AES_ENCRYPT_128>
11  mov    %rax,-0x10(%rbp)
12  movdqu %xmm15,-0x18(%rbp)
13  ......
```

Code 8: Function Prologue of Compiler based P-SSP-OWF

```
1   movq   -0x10(%rbp),%xmm15
2   movhps 0x8(%rbp),%xmm15
3   movq   %r13,%xmm1
4   punpckhdq %r12,%xmm1
5   callq  <AES_ENCRYPT_128>
6   comiss -0x18(%rbp),%xmm15
7   je     Label
8   callq  <__stack_chk_fail@plt>
9   Label
10  leaveq
11  retq
```

Code 9: Function Epilogue of Compiler based P-SSP-OWF

In the function epilogue, we load both of CPU cycle data and return address, re-encrypt and compare result with ciphertext saved on stack. Any modification of return address, CPU cycle data and ciphertext will lead a mismatch and be detected.

Security of P-SSP-OWF Time Stamp Counter is unique and linear increasing with the execution of program, however, the AES encryption is not a linear function and the key in `r12` and `r13` is randomly generated. So, the canaries in the stack are unique and unknown for every stack frame. These features ensure that P-SSP-OWF has the following three effectiveness: firstly, brute force attack is infeasible; secondly, even if the attacker get canaries in the stack, it can not figure out the key because AES is secure against known-plaintext attack; thirdly, since attacker can not broken the AES key in registers, it can not successfully construct exploiting script with expectant return addresses (to hijack control flow) and corresponding ciphertext.

At last, we emphasize that P-SSP-OWF is different from return address encryption protections such as RAP [13] and PointGuard [14]. Encryption in P-SSP-OWF aims to implement a dynamic and polymorphic canary, which is unpredictable and imponderable for attackers. Encryption of Time Stamp Counter can achieve our goals. Besides, we use return address as part of plaintext because the shortest encryption length supported by AES-IN is 128-bit and consequently we make full use of redundant 64-bit to hold return address. For return address protections When compare P-SSP-OWF with return address encryption (RAP, PointGuard), which hide and protect return addresses via XOR encryption, we find that P-SSP-OWF is more powerful. Return address encryption can not prevent brute force attack and memory disclosure. We acknowledge that those effectiveness is result of the adoption of more secure encryption methods, AES-IN.

VI. Evaluation

In this section we evaluate the performance overhead of P-SSP and its three extensions. Our platform is a PC with 24 GB of main memory, an Intel(R) Core(TM) CPU i7-4770k processor with 128KB L1, 1MB L2, and 8MB L3 cache. The operating system is Debian 8.7.1 with a 3.16.0-4-amd64 Linux kernel.

A. Performance

We measure several types of overheads incurred by P-SSP, including CPU time, code expansion and memory usage. To assess its overall impact upon network services, we also assess the increment of the service response time. The applications we use include the SPEC CPU®2006 [15], [16], Apache2, Nginx, MySQL, and SQLite. The latter four are multithreaded programs running as network servers.

1) Runtime Performance: The industry-standard SPEC CPU®2006 benchmark suite includes SPECint® [15] benchmark suite for integer operations and SPECfp® [16] benchmark suite for floating point operations. We compile the source code of the benchmark programs with the P-SSP option. The runtime environment is configured by applying `"submit = LD_PRELOAD=[path_to_preload_lib]/libpoly_canary.so $command"` and `"use_submit_for_speed=1"` to the config file of the benchmark. Native executions of the benchmarks are performed with the default compilation option and configuration.

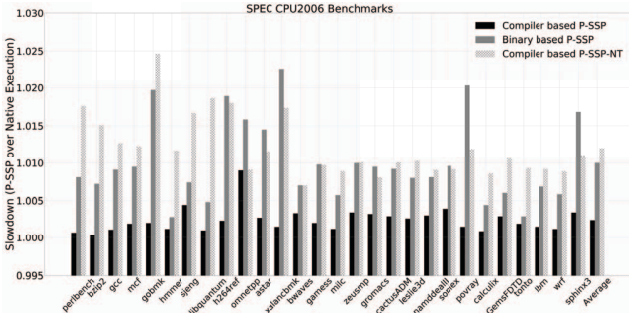


Fig. 5: Runtime Overhead of P-SSP Against Native Executions Using SPEC CPU2006 Benchmark Suit

Figure 5 summarizes the performance overhead of compiler based and instrumentation based P-SSP over native executions. We find that our binary rewriter tools for dynamic linking program and static linking program have similar runtime performance. The compiler based P-SSP incurs 0.24% slowdown in average, while the instrumentation based P-SSP incurs 1.01% slowdown in average. As compared to Table I, it is evident from our experiment results that P-SSP attains the security RAF [3], DynaGuard [4] and DCR [3] with a much lighter overhead in both compiler and instrumentation implementations.

2) Code Expansion: P-SSP uses more instructions than SSP. We therefore also evaluate the code expansion over the native code. We use the 28 programs in SPEC CPU2006 benchmarks for evaluation. We measure the size of the binaries compiled with the P-SSP option and the size of the instrumented binaries, and compare them against the native code size compiled with the default options.

Table II summarizes the code expansion rates. The compiler based version increases the code size by 0.27% while the instrumentation based version for dynamic linked executable has code expansion. Static linked binaries under P-SSP instrumentation grows about 2.78%, attributing to the two new glibc functions, `__stack_chk_fail()` and `fork()`. We remark that static link glibc binaries are not popular. Among 42383 ELF files in Ubuntu and 44837 ELF files in Debian, only 2 static linked files are found.

TABLE II: Code Expansion Rate by Different P-SSP Implementation

Compilation	Instrumentation (dynamic link)	Instrumentation (static link)
0.27%	0	2.78%

3) Response Performance: We run Apache2, Nginx, MySQL, and SQLite in the multithread mode to evaluate P-SSP's overall performance toll on these services. Apache2 and Nginx are stressed by running Apache Benchmark [17] with 100,000 requests and concurrency degree set as 500. We measure their average response time under different settings: native execution (compiled with default options), compiled with P-SSP, instrumented with P-SSP. The results are reported in Table III. The database services, i.e. MySQL and SQLite, are benchmarked by using the *sysbench* application and *threadtest3.c* [18], respectively. We measure the average query execution time and their memory usages. The results are reported in Table IV.

Both Table III and IV show that P-SSP incurs negligible overhead to the performance of web and database servers. The main reason is that the computation overhead of TLS canary generation and canary checking in function epilogues only account for a tiny portion of the entire web or database transactions. Therefore, the overall performance loss is not significant.

B. P-SSP v.s. Extensions

P-SSP extensions uses cryptographic instructions and more complicate logics in the function prologue and epilogue. Both P-SSP-NT and P-SSP-LV use the `rdrand` instruction to generate random numbers while P-SSP-OWF uses AES instructions. As these cryptographic operations consume more CPU cycles than memory copying and exclusive-OR operations, we run experiments to measure their overhead.

TABLE V: Average of CPU cycles spent by the function prologue and epilogue for P-SSP and its three extensions

P-SSP	P-SSP-NT	P-SSP-LV		P-SSP-OWF
		2 variables	4 variables	
6	343	343	986	278

As shown in the Table V, the `rdrand` instruction used for random number generation in P-SSP-NT and P-SSP-LV costs about 340 more CPU cycles, which is roughly 97 nanoseconds. The AES operations in P-SSP-OWF costs about 272 more CPU cycles, which is merely 77 nanoseconds. The experiments of P-SS-LV with two variables report similar results as in P-SSP-NT, because only one random number is generated in both settings. For the similar reason, experiments in P-SSP-LV with four variables generate three random numbers and hence report nearly three times more CPU cycles. We argue

TABLE III: P-SSP's Performance Impact on Web Servers (average time per request in milliseconds)

	Native Execution	**Compiler based P-SSP**	**Instrumentation based P-SSP**
Apache2	33.006	33.008	33.099
Nginx	3.088	3.090	3.088

TABLE IV: P-SSP's Performance Impact on Database Servers

	Native Execution		**Compiler based P-SSP**		**Binary based P-SSP**	
	Query Execution(ms)	Mem Usage(MB)	Speed	Mem Usage(MB)	Query Execution(ms)	Mem Usage(MB)
MySQL	3.33	22.59	3.33	22.59	3.33	22.59
SQLite	167.27	20.58	167.27	20.58	167	20.58

that it seems to be affordable for programs to adopt P-SSP-NT and P-SSP-OWF, as the incurred CPU time overhead is negligible as compared to the time for the entire tasks which is often measured in the order of milliseconds.

C. Compatibility & Effectiveness

We also run two sets of experiments to test the compatibility between P-SSP and SSP. In the first set of experiments, we compile SPEC CPU2006 benchmark suites with P-SSP option, while the glibc libraries are compiled with the default SSP option. In the second set of experiments, we compile the glibc libraries with P-SSP option and compile the benchmark programs with the default option. In both cases, the binaries have a mixture of P-SSP and SSP. Our experiments corroborate our analysis that P-SSP is fully compatible with SSP. The benchmark programs behaves normally without complaining any error. No false positive occurs when the child process returns to the stack frames inherited from the parent process.

To test the effectiveness of P-SSP, we run the byte-by-byte attacks on Nginx and Ali compiled with SSP and P-SSP options. The attacks are successful upon SSP-compiled Nginx and Ali. However, the same attack script have failed when attack the P-SSP compiled version.

VII. Discussion & Future Work

A. Memory Corruption Attacks

The key value of P-SSP and its variants is that they eliminate a widely used attack vector. We emphasize that P-SSP can not address all kinds of memory corruption attacks. For instance, the format string attack can read or write canary value directly and therefore bypasses the canary checking. We refer readers to [7] which systemizes the knowledge of memory corruption attacks and countermeasures. It remains as an open problem to develop an efficient and practical scheme to cope with the full spectrum of memory corruption attacks. It is more promising to integrate different types of countermeasures, such as Non-executable stack (DEP) [19], ASLR [20], CFI [9], and stack canary, so that the attack surface is minimized.

B. Comparision with DEP, ASLR and CFI

Non-executable stack stops code-injection, and does not prevent return address manipulation to change the control flow (e.g., in order to bypass an authentication checking). P-SSP detects illegal return address when the function returns. Since code-injection in the stack does not involve function epilogue, P-SSP cannot detect such attacks. It is more difficult for integrity protection as it cannot be enforced by using the MMU.

ASLR aims to hide the address space layout. P-SSP is to maintain the integrity of critical data in stack, including return addresses and local variables. A secure ASLR surely increases the difficulty of breaking P-SSP. Unfortunately, commodity OSes only use coarse-grained ASLR which can be easily broken.

CFI and P-SSP overlap in terms of their functionalities, as the main purpose of P-SSP is to detect illegal return addresses, which can also be detected by CFI. However, CFI cannot protect local variables in the stack.

C. Stack Layout and Canary Length

P-SSP places two memory words in the stack as the canary whose size is therefore twice of the SSP canary's. As a result, the stack layout of P-SSP is different from the one for SSP. Although the layout change does not cause any issue for binaries compiled with P-SSP, it affects instrumentation based P-SSP. As shown in Section V-C, we have to sacrifice security to overcome the challenge by having the size of the P-SSP stack canary.

One might suggest to place C_0 in the TLS as the TLS shadow canary and compute C_1 in every function prologue so that only C_1 is used as the stack canary. The function epilogue then tests whether $C_1 \oplus C_0 \oplus \mathbf{C}$ equals to 0. Since C_1 has the same length as $\mathbf{C}$ in the TLS, this method preserves the stack layout. Unfortunately, it is not satisfactory as it seems because it gives rise to another problem. When a process forks out a child process, the child's new C_0 replaces its parent's C_0 in the TLS. Hence, when the control flow of the child returns to its parent's code using stack frames created before forking, the parent's epilogue function does not have the proper TLS shadow canary (i.e. C_0) to check and the program is doomed to crash.

Hence, we propose a different method to preserve the 64-bit canary length. As showed in Figure 6, we allocate a global buffer for each thread (Linux process) to hold half of canaries corresponding to the other half of canaries in stack. When C_0 is pushed into the stack frame, we place the corresponding C_1

into the global buffer. Once `fork()` is invoked to generate a children process, the child processes clones their parent process' global buffer.

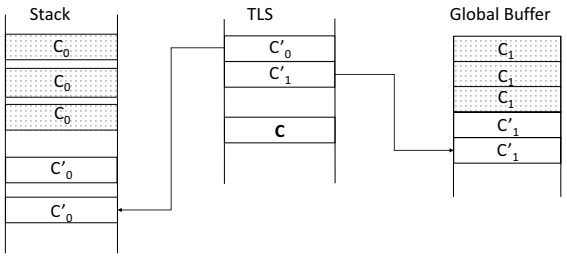


Fig. 6: Use a global buffer to offer the additional storage for P-SSP. The shadowed region denotes the canaries created by the current process's parent process.

VIII. Conclusion

Stack smashing protection is a popular compiler-based technique for detecting and defeating stack overflow attacks. However, it is known to be vulnerable to the byte-by-byte attack. In this paper, we propose P-SSP where the stack canary is properly re-randomized without affecting the TLS canary. Hence, the adversary's advantage acquired via partially revealing a canary cannot be accumulated. P-SSP is fully compatible with SSP and it does not require stack frame tracing as used by existing schemes. We also make three extensions of P-SSP, namely P-SSP-NT, P-SSP-LV and P-SSP-OWF. P-SSP-NT update the stack canary for each frame and does not change the existing TLS; P-SSP-LV widens the protection coverage from the return address to local variables; P-SSP-OWF uses a one-way function to attain resilience to memory exposure so that the canary leakage in one stack frame does not affect other frames.

We have built an LLVM plugin to emit function prologues and epilogues using P-SSP, and a binary rewriting tool to instrument legacy binaries for P-SSP. The benchmark programs show that P-SSP incurs non-significant runtime overhead. We have also implemented LLVM plugins for P-SSP-NT and P-SSP-OWF, and run experiments to compare the CPU time overheads between P-SSP and three extensions. The results show that it is promising to deploy these extensions attributing to the recent hardware support for cryptographic operations. Our implementation of P-SSP is available at: https://github.com/zhilongwang/PolymorphicCanaries.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their useful comments and suggestions. We would like to thank Petsios et al. for their help when evaluating P-SSP with the SPEC CPU 2006 benchmark. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078, 61073027, 90818022, and 61321491), and the Chinese National 863 High-Tech Program (2011AA01A202). This research was supported, in part, by the Singapore National Research Foundation under the NCR Award: NRF2014NCR-NCR001-012.

References

[1] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[3] H. Marco-Gisbert and I. Ripoll, "Preventing brute force attacks against stack canary protection on networking servers," in *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*. IEEE, 2013, pp. 243–250.

[4] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 351–360.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[6] W. H. Hawkins, J. D. Hiser, and J. W. Davidson, "Dynamic canary randomization for improved software security," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. ACM, 2016, p. 9.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[8] T.-c. Chiueh and F.-H. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Distributed Computing Systems, 2001. 21st International Conference on.* IEEE, 2001, pp. 409–417.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[10] A. Zúquete, "Stackfences: a run-time approach for detecting stack overflows," *Electrónica e Telecomunicações*, vol. 4, no. 3, pp. 403–412, 2004.

[11] LLVM, Available: http://llvm.org/.

[12] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[13] B. S. . T. P. Team, "Return address protection," Available: https://grsecurity.net/rap_announce2.php.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, 2003, pp. 91–104.

[15] "SPEC CPU 2006 int benchmark," Available: https://www.spec.org/cpu2006/CINT2006/, Standard Performance Evaluation Corporation.

[16] "SPEC CPU 2006 float benchmark," Available: https://www.spec.org/cpu2006/CFP2006/, Standard Performance Evaluation Corporation.

[17] "Apache http server benchmarking tool," Available: https://httpd.apache.org/docs/2.4/programs/ab.html, The Apache Software Foundation.

[18] S. Developers, "How sqlite is tested," Available: https://www.sqlite.org/testing.html.

[19] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[20] P. Team, "Pax address space layout randomization (aslr)," 2003.