DynaGuard: Armoring Canary-Based Protections against Brute-force Attacks

Theofilos Petsios,
Angelos D. Keromytis

Vasileios P. Kemerlis

Michalis Polychronakis

Columbia University

Brown University

Stony Brook University

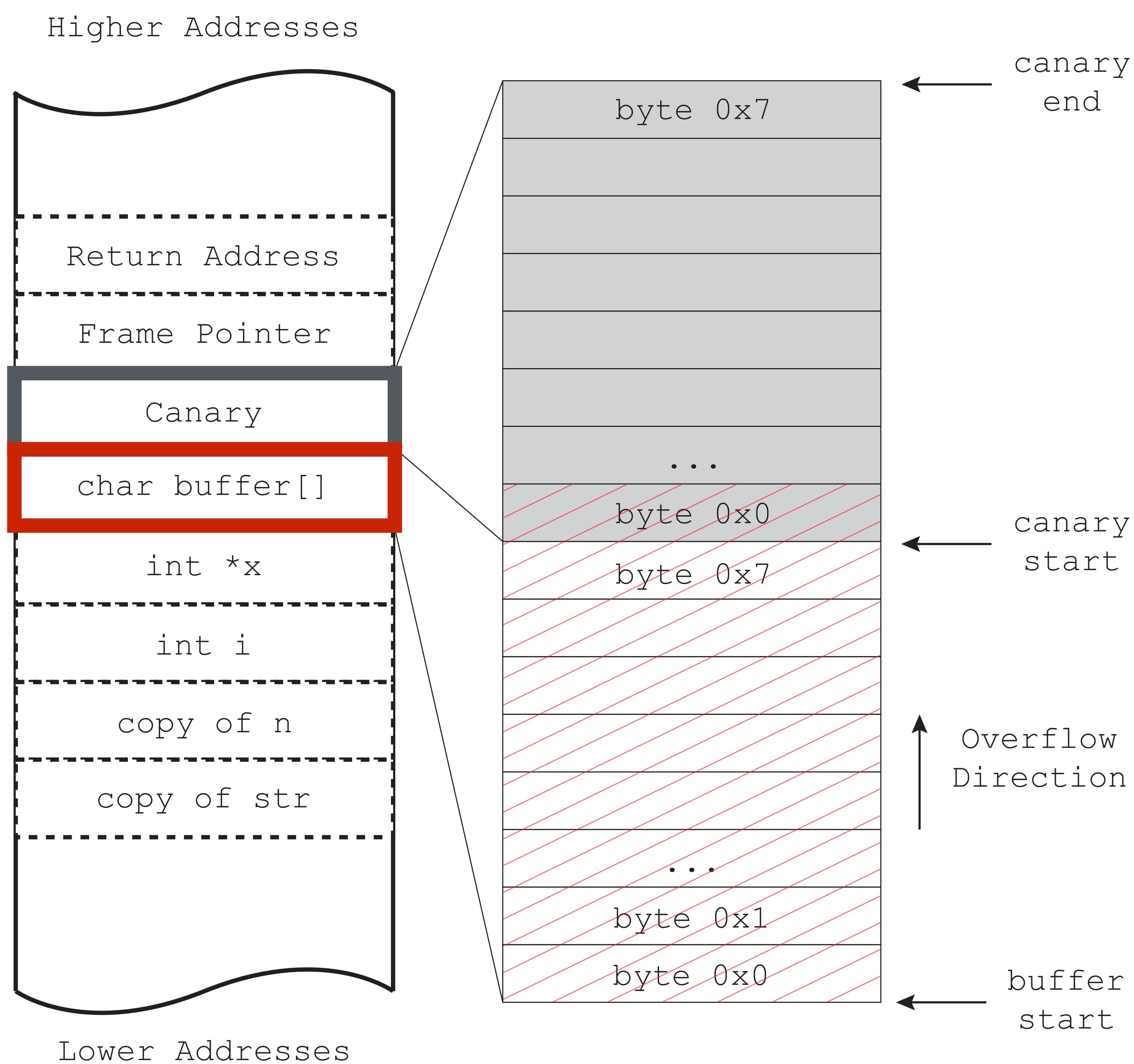
2015 Annual Computer Security Applications Conference (ACSAC)
Los Angeles, California, USA

Background: Stack Smashing Protection

- Prevents the **overwrite** of the return address by a stack buffer overflow
- Places a random value after **critical** data in the stack
  - Random value: ➡ “Canary” or “Canary Cookie”
  - Critical data ➡ Return address, Frame pointer, etc.
  - The canary is 4 bytes long in x86, 8 bytes in x86-64
- Generated dynamically at the creation of each thread, and stored in the Thread-Local Storage (TLS) area
- Checked upon function epilogue
- Supported in GCC, Microsoft VS (/GS) and LLVM

Background: Stack Smashing Protection

```
int vuln(int n, char *str)
{

   int i;
   int *x = NULL;
   char buffer[8];

   ...

   /* unbounded copy */
   memcpy(buffer, str, n);

   ...

}
```



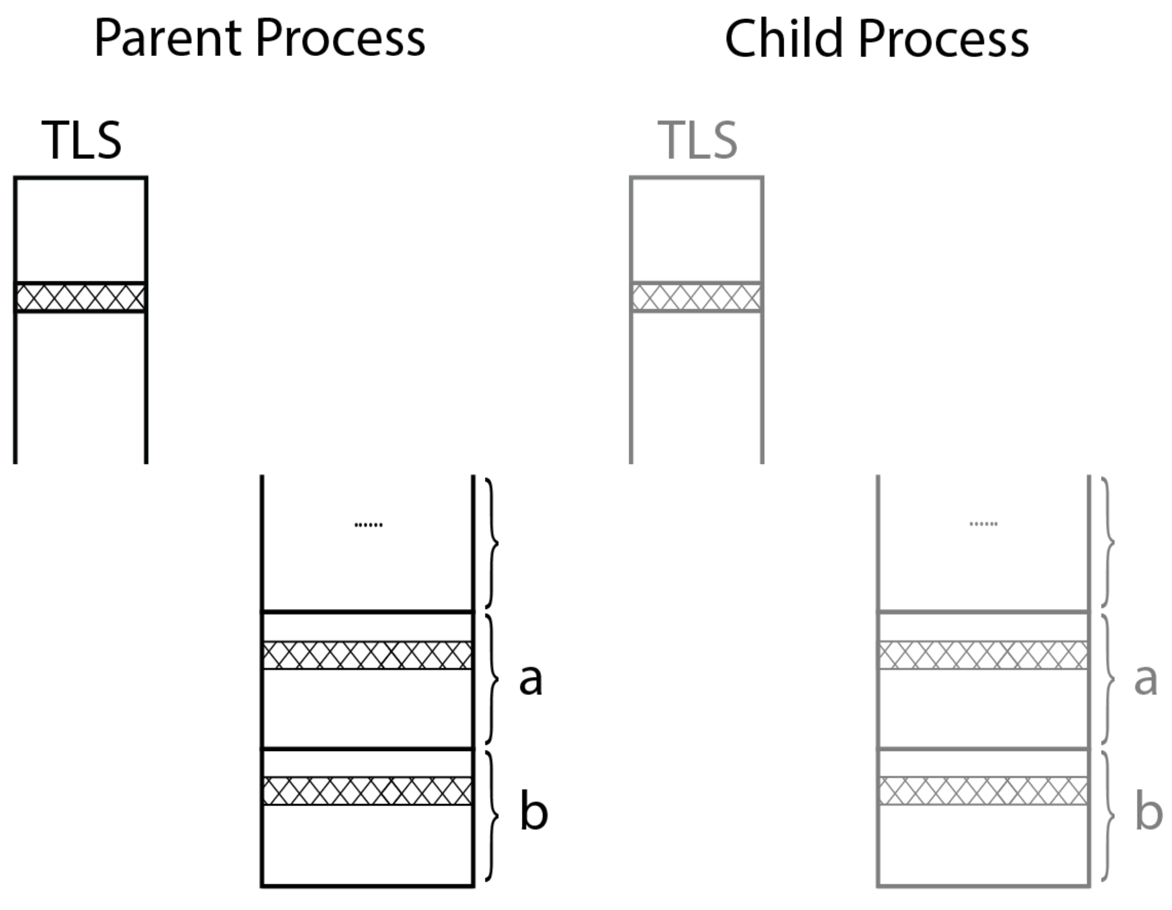
Canary Brute-force

An attacker may brute-force the canary **byte-by-byte** in very few attempts if they are able to perform the following steps:

- Force child processes to be forked by the same parent process
- Verify if these child processes crashed or not
- Overwrite a single byte of the canary each time until all the bytes are recovered

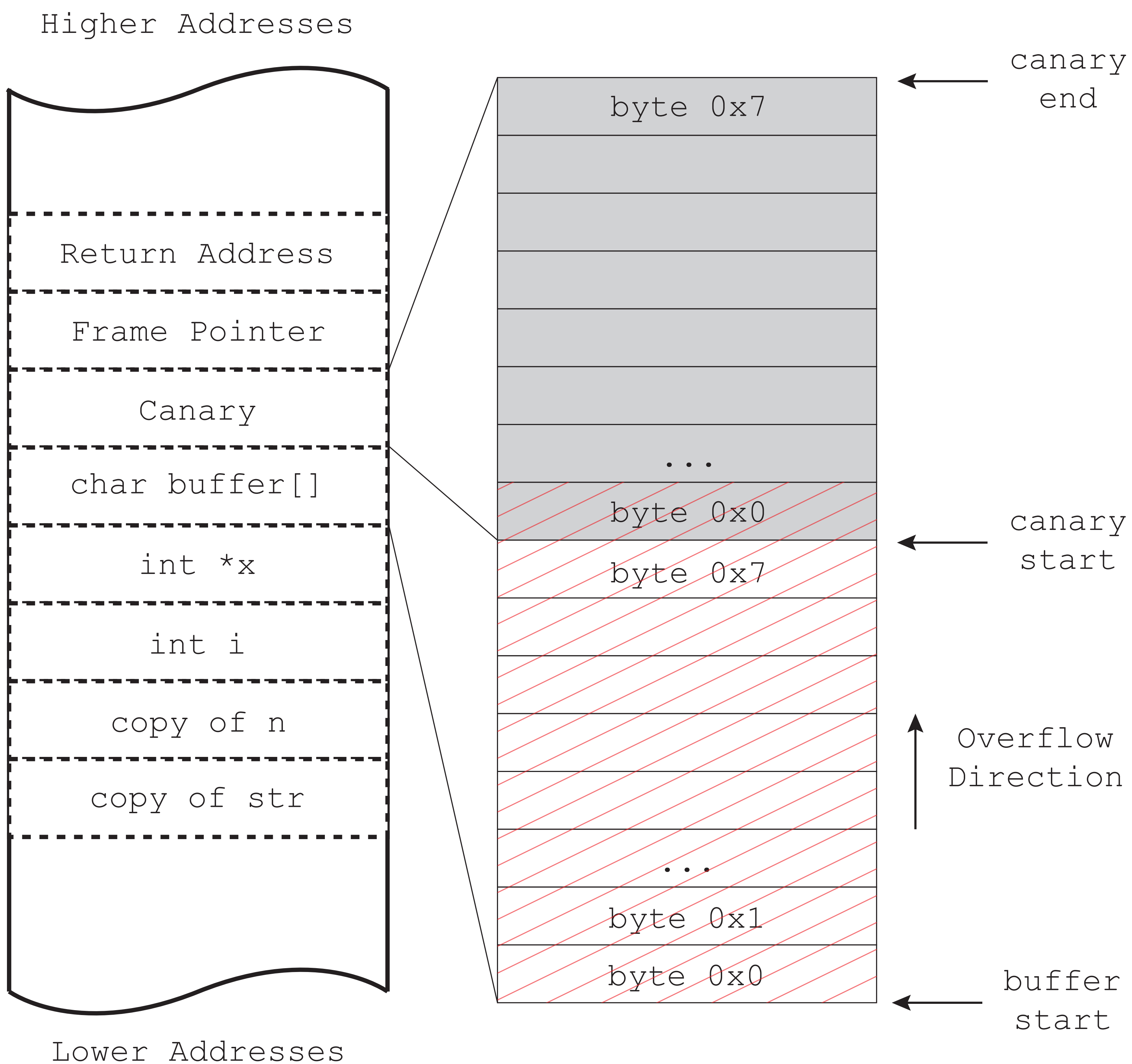
Canary Brute-force

- Possible due to the current process creation mechanism:

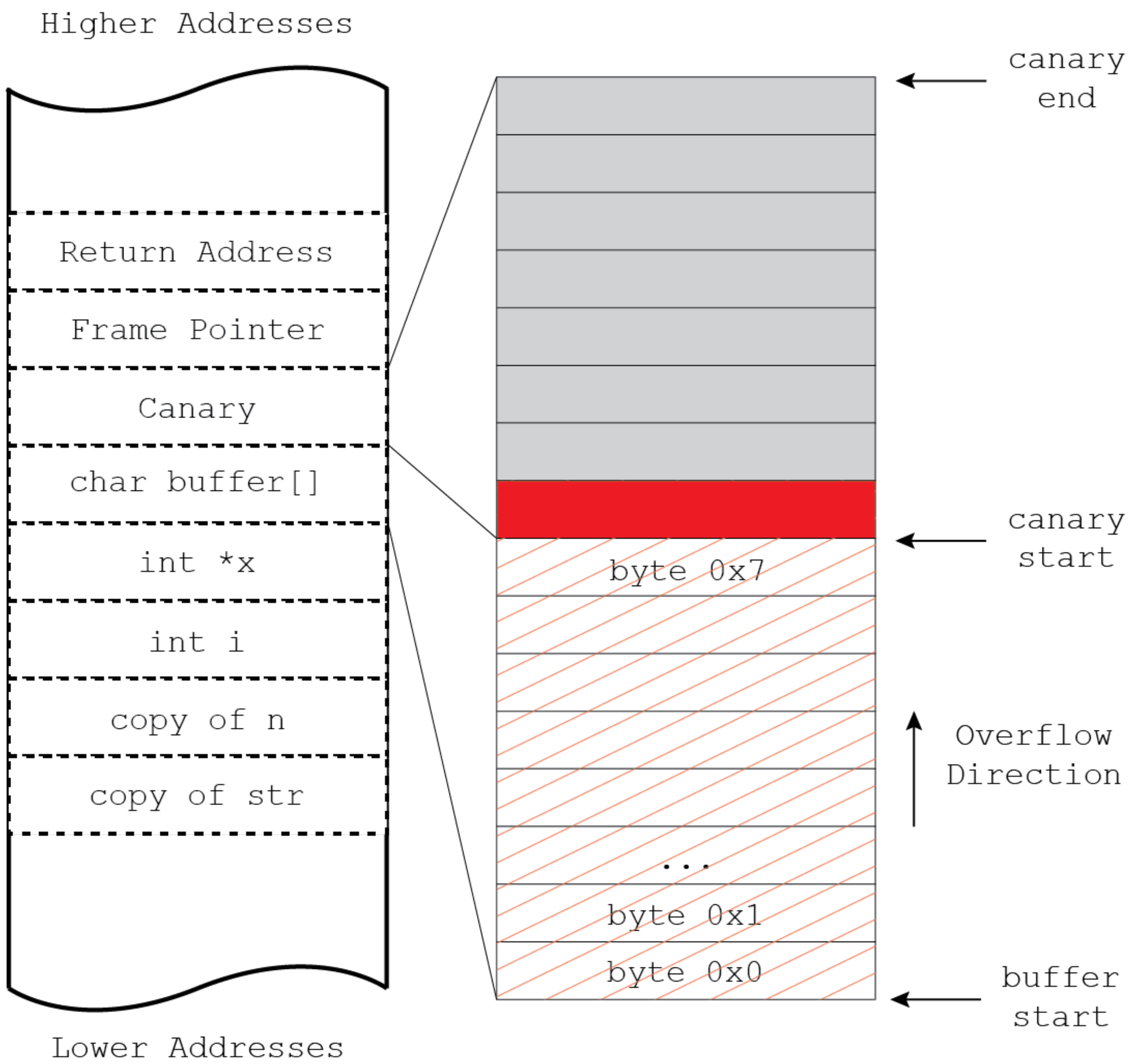


- Certain data is inherited from the parent process, although it should be different (other examples include VM side channel attacks and the PRNG state in forked processes)

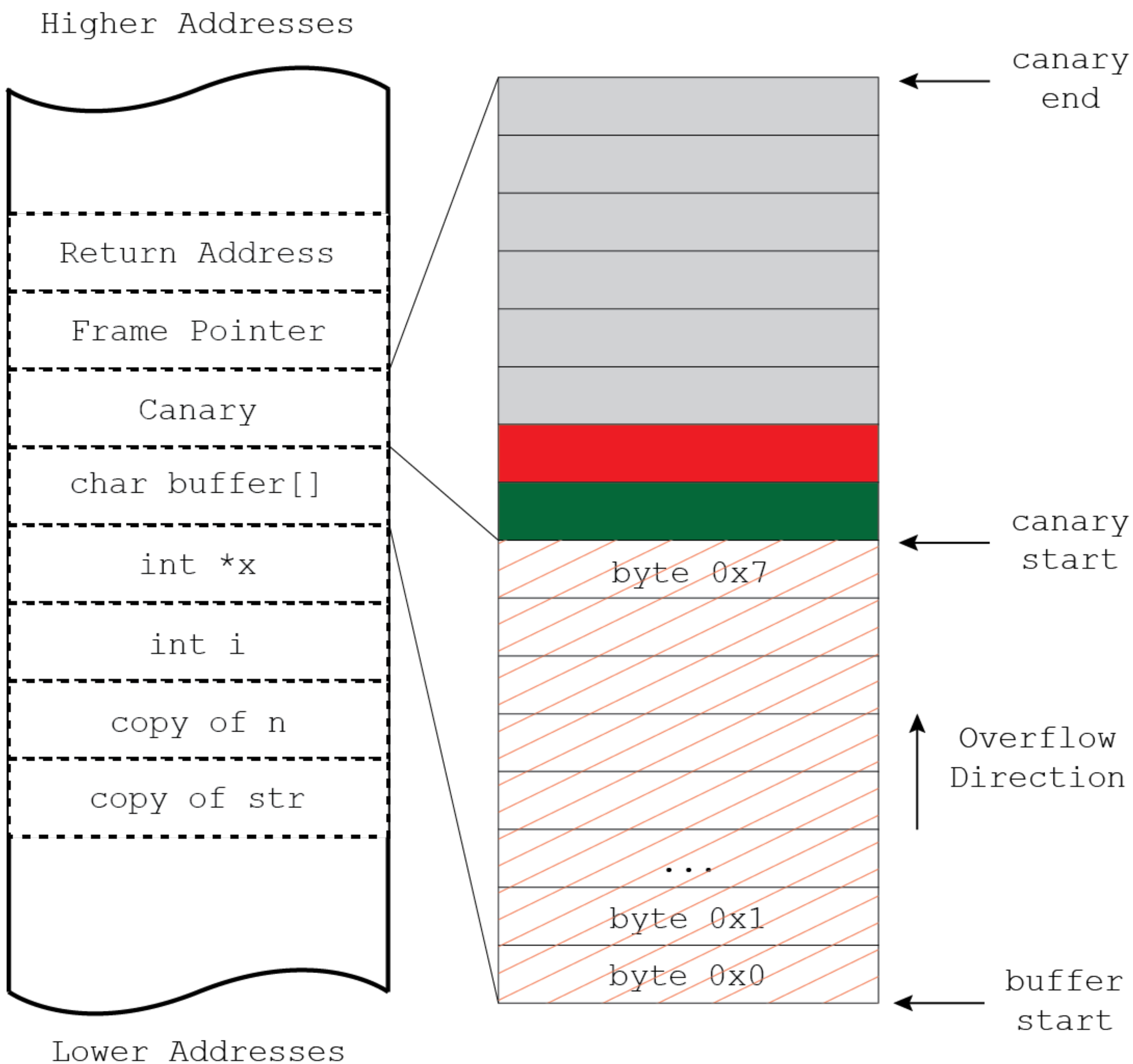
Canary Brute-force



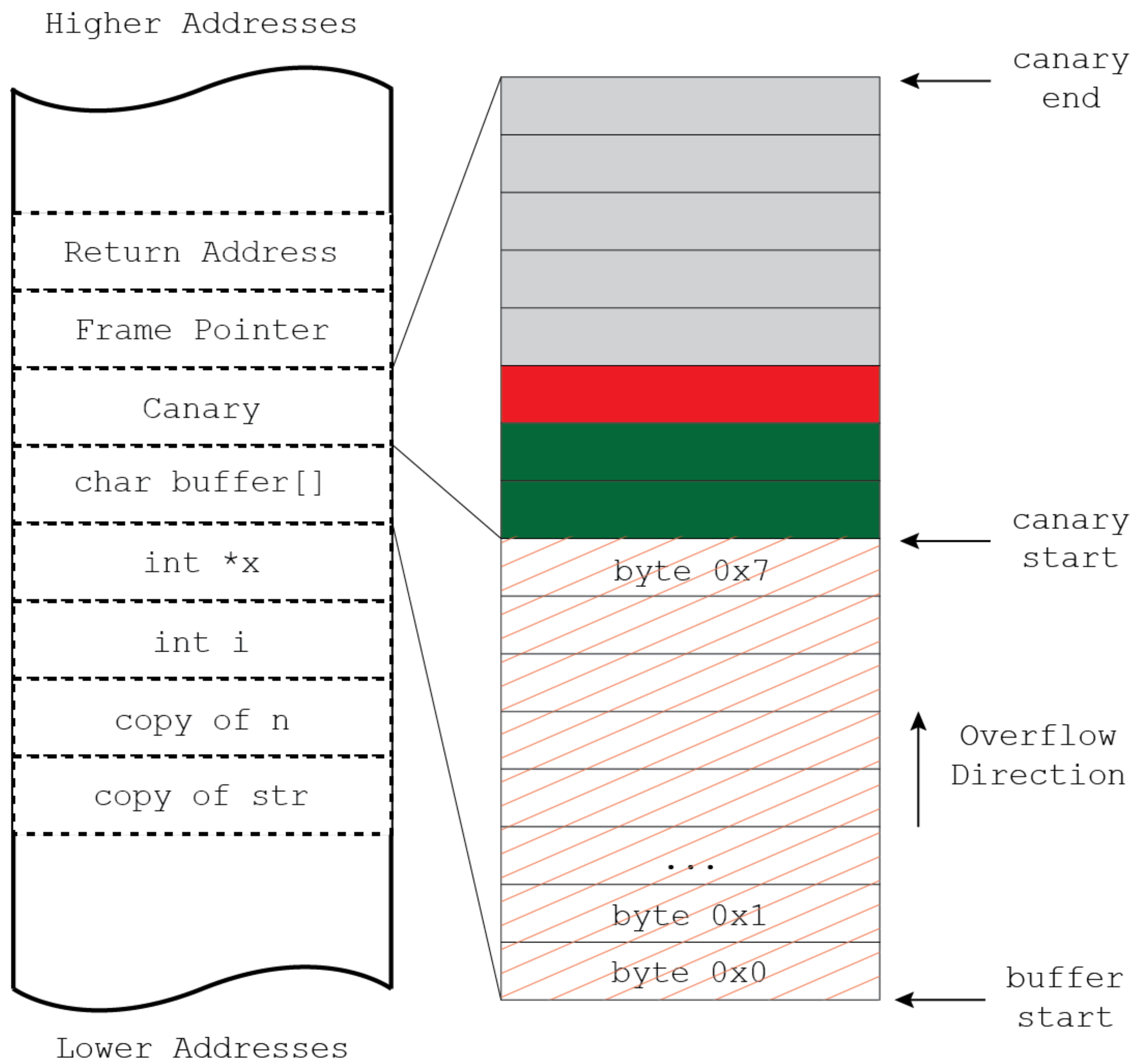
Canary Brute-force



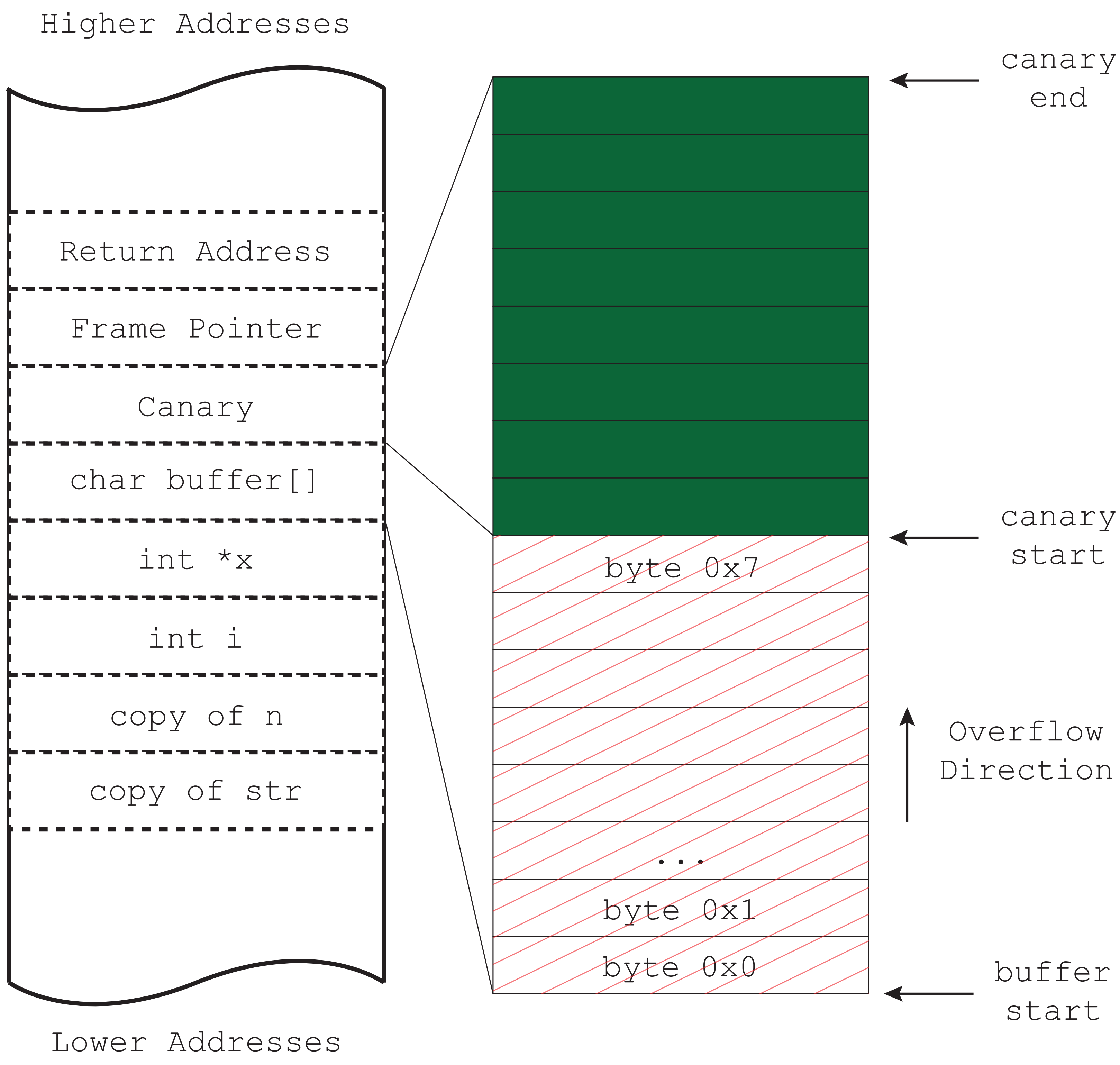
Canary Brute-force



Canary Brute-force



Canary Brute-force



A byte-by-byte brute-force requires 4*256 = 1024 attempts on average on x86 and 2048 on x86-64, assuming a fully random canary

Canary Brute-force Guessing Timeline

- **2006** — Ben Hawkes introduced the technique in RUXCON 2006 (Title: "Exploiting OpenBSD")
- **2010** — Adam Zabrocki (pi3) discussed remote stack exploitation techniques in Linux, FreeBSD and OpenBSD and among other things, revisited Ben's attack in Phrack #67
- **2013** — Nikolaos Rangos (Kingcope) released an exploit for the Nginx web-server that builds upon the previous attack(s) to construct a remote exploit
- **2014** — Andrea Bittau et al. introduced the BROP technique, which among other things, uses a generalized version of the above to leak/bypass stack canaries

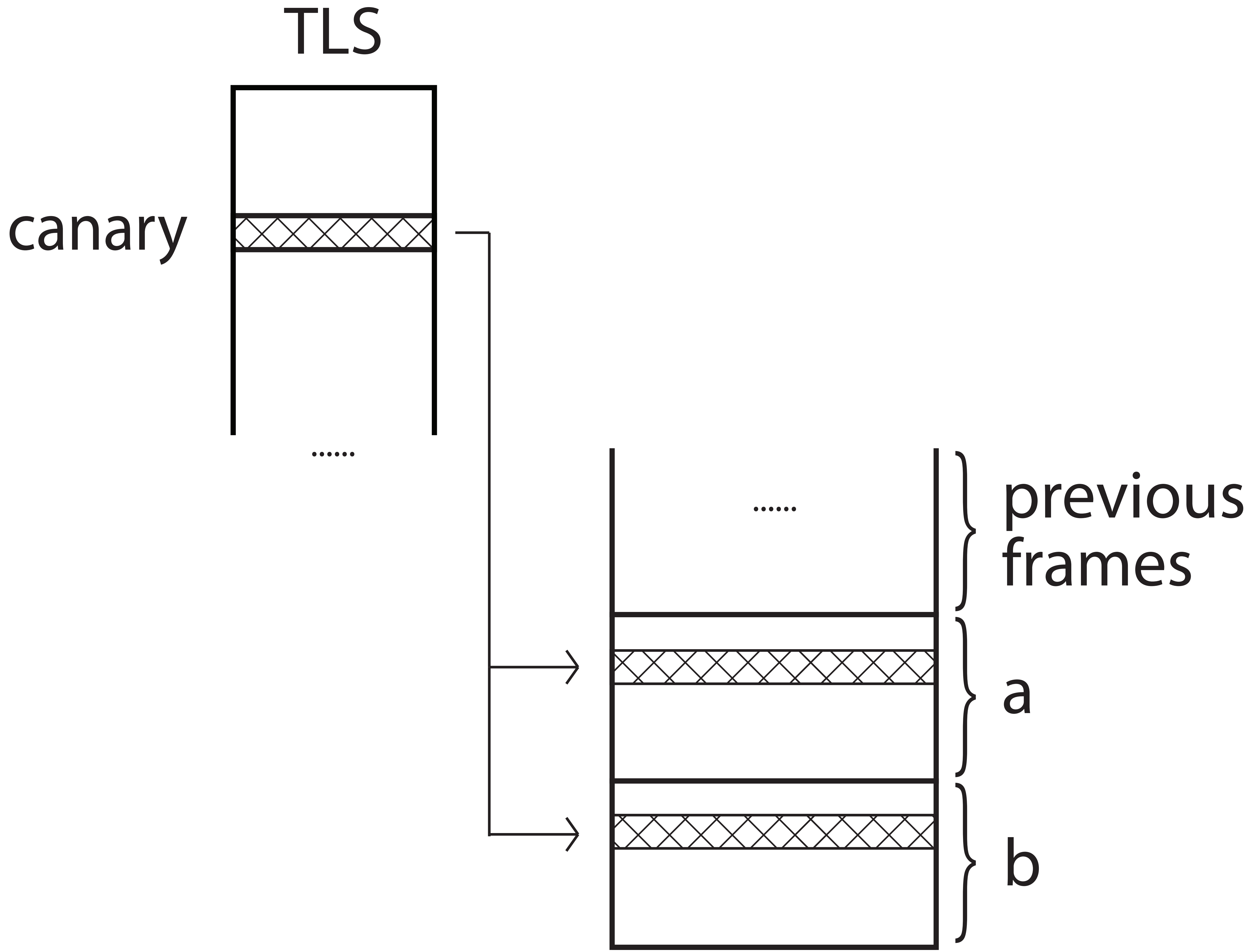
DynaGuard Design

Key idea: Upon each `fork()` update the inherited (old) canaries in the child process

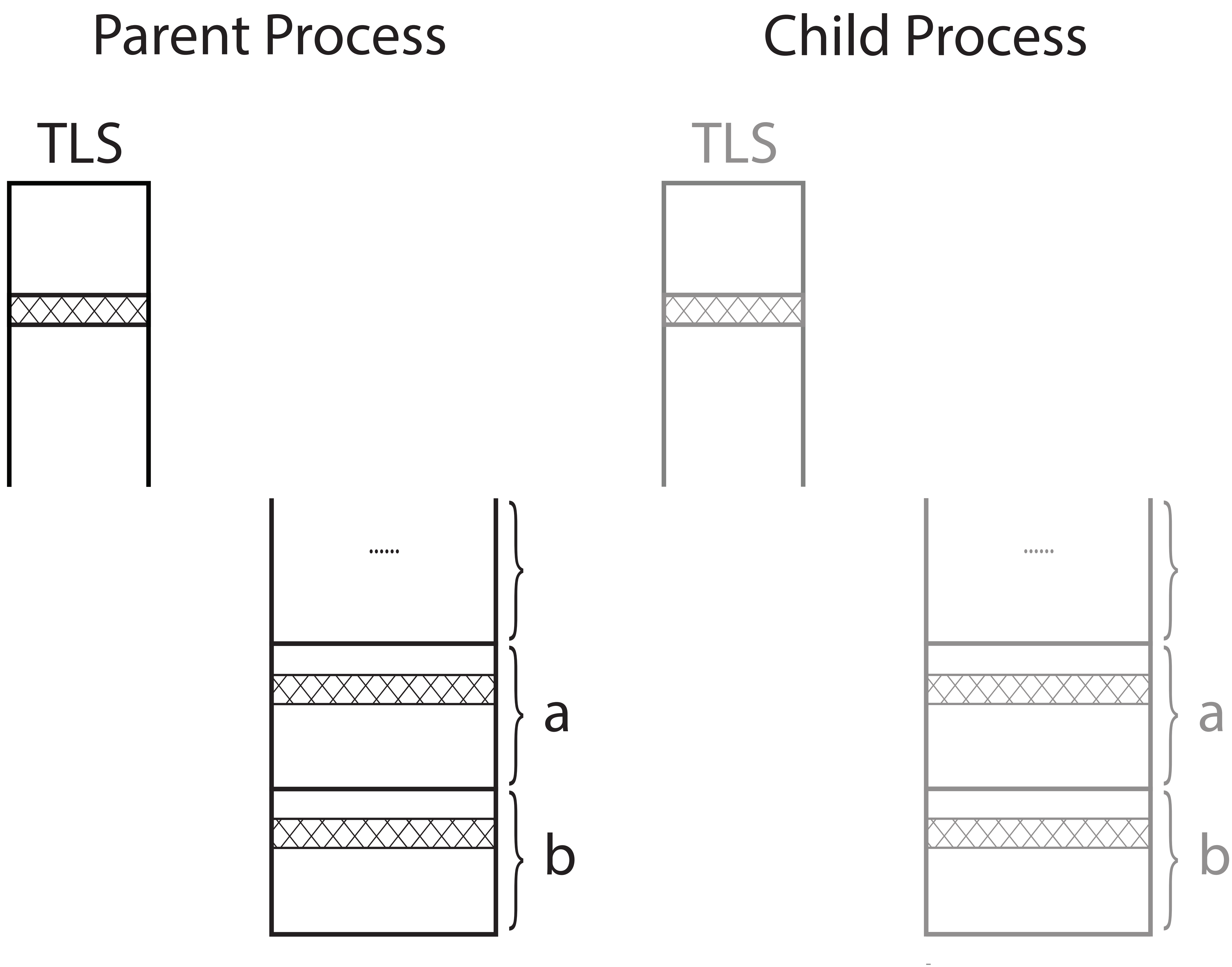
- Update the canary in the TLS of the new (child) process
- Update the canaries in all inherited stack frames (from the parent process) with the new canary value

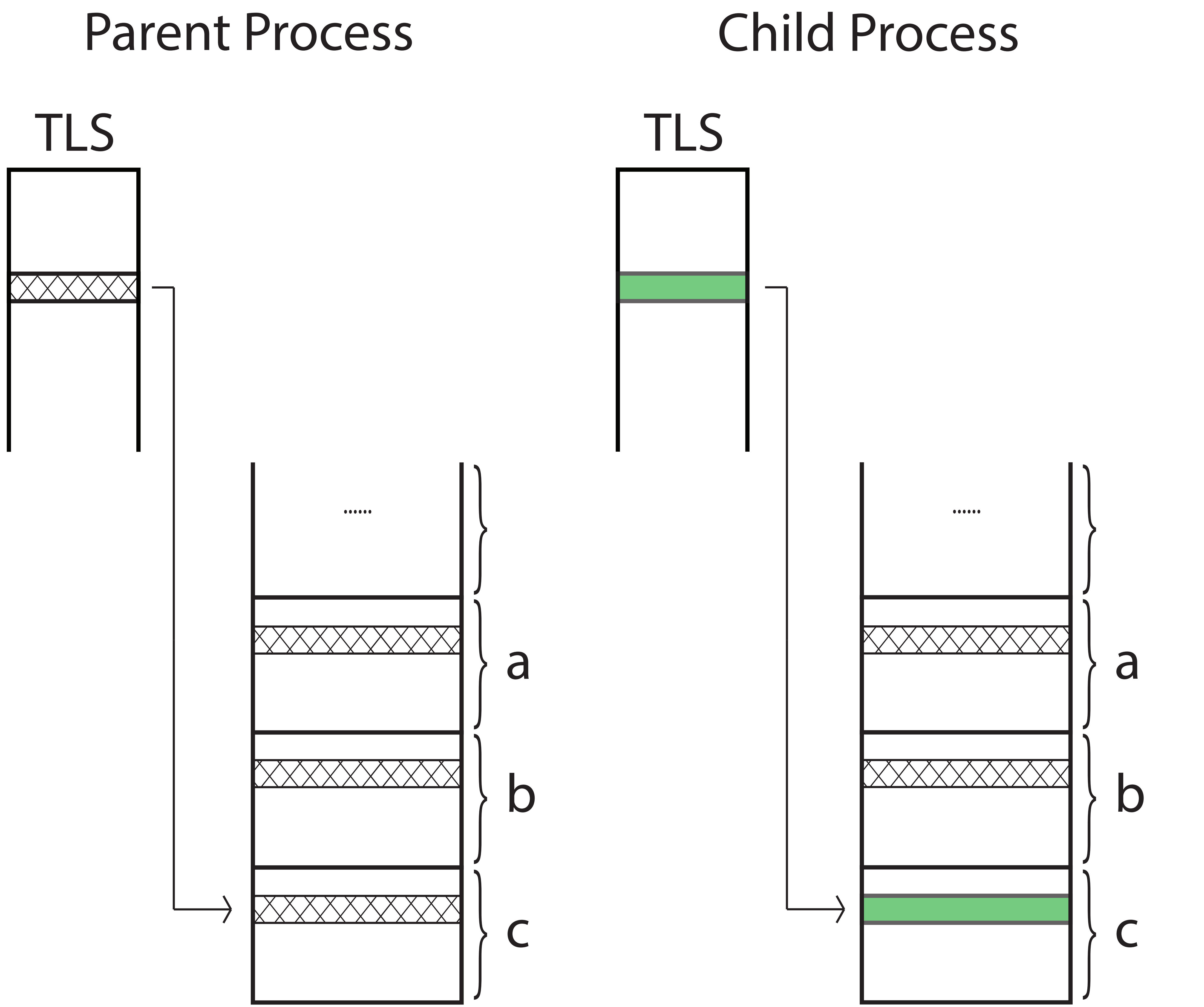
Simply updating the canary in the TLS* for new (child) processes is not enough as it will cause a false abort if execution reaches one of the parent's inherited frames

*as proposed in a recent paper

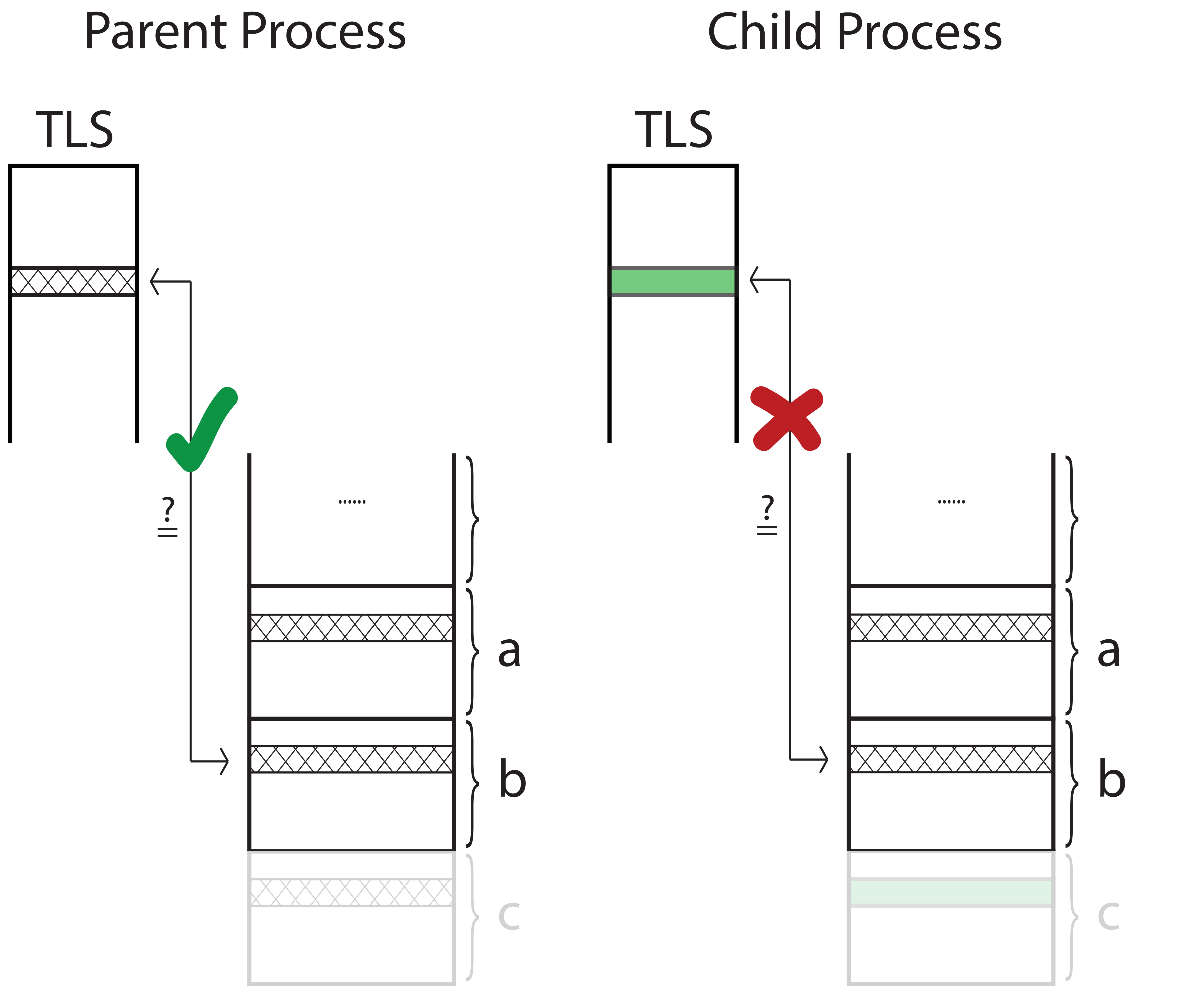


TLS
canary
......
......
previous frames
a
b

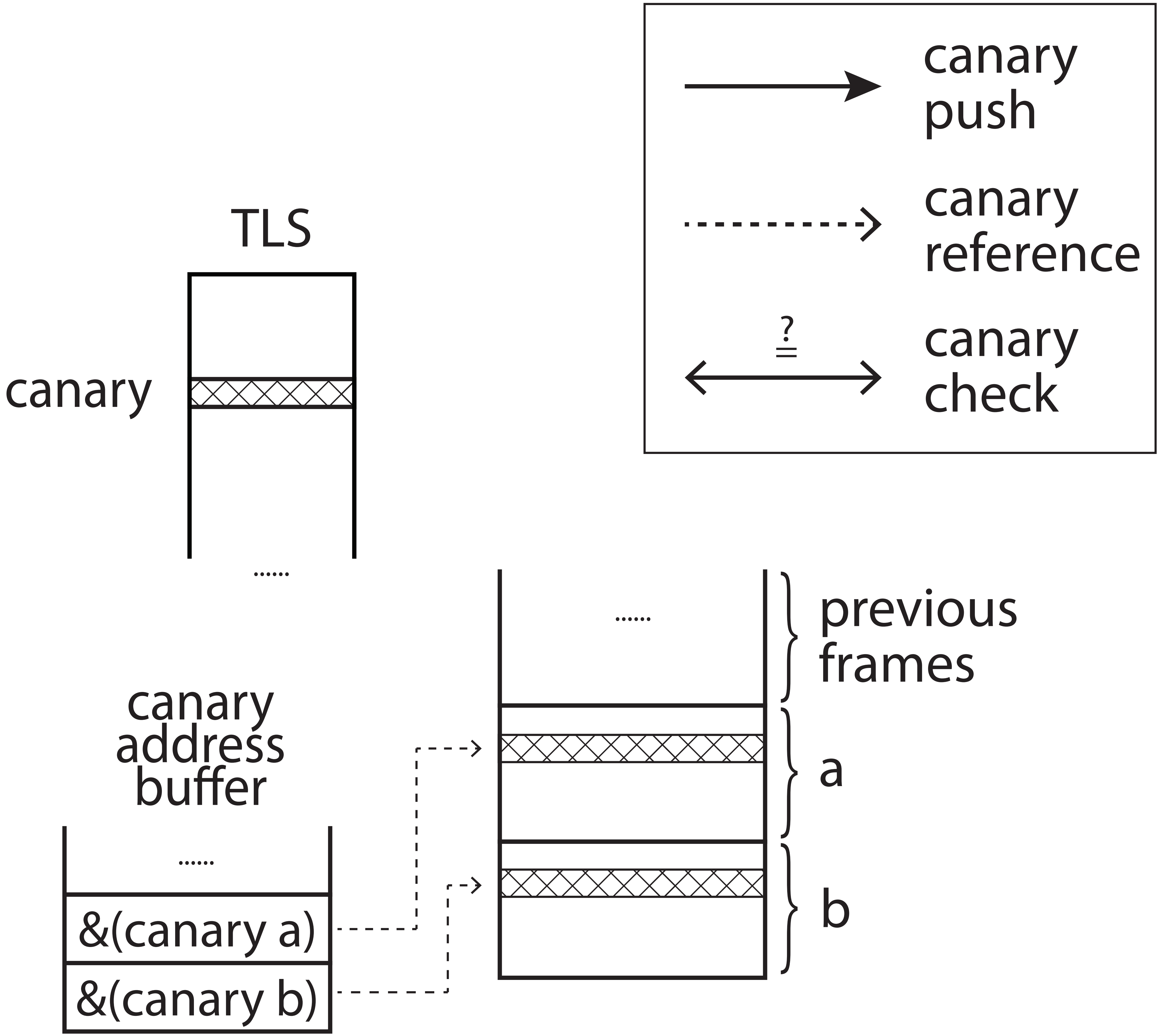

Parent Process
Child Process
TLS
TLS
......
a
b
......
a
b

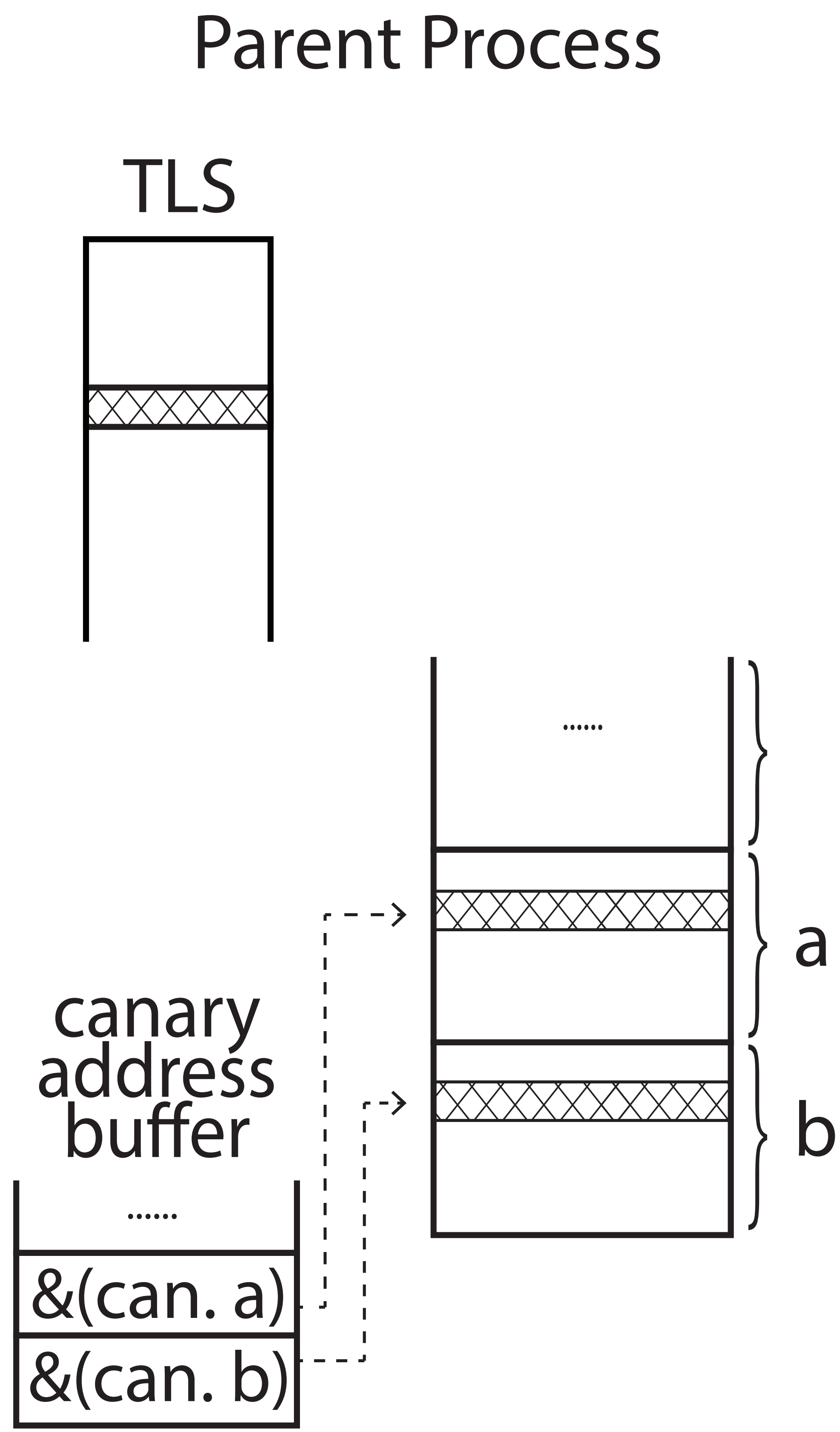


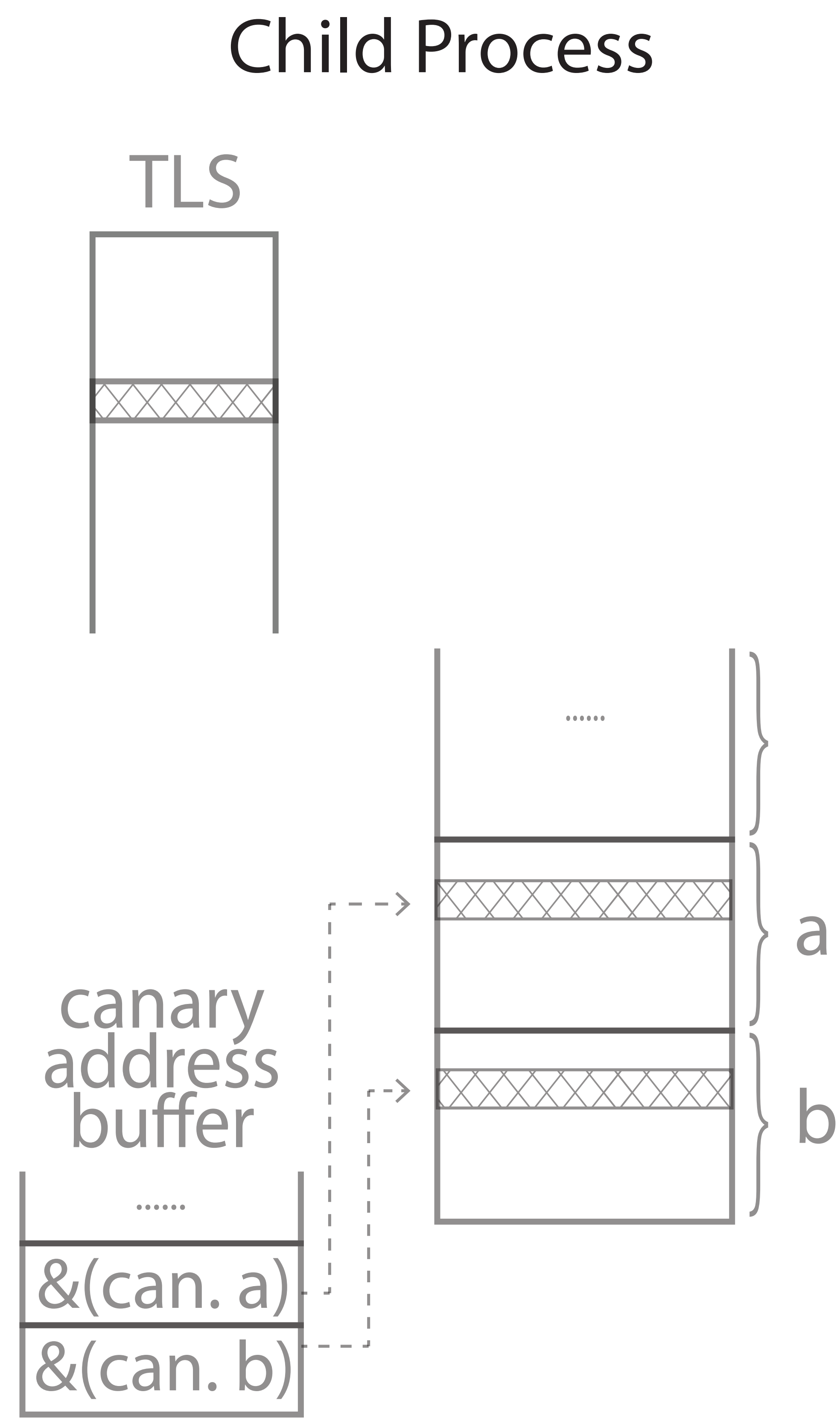
Parent Process
Child Process
TLS
TLS
......
......
a
b
c
a
b
c

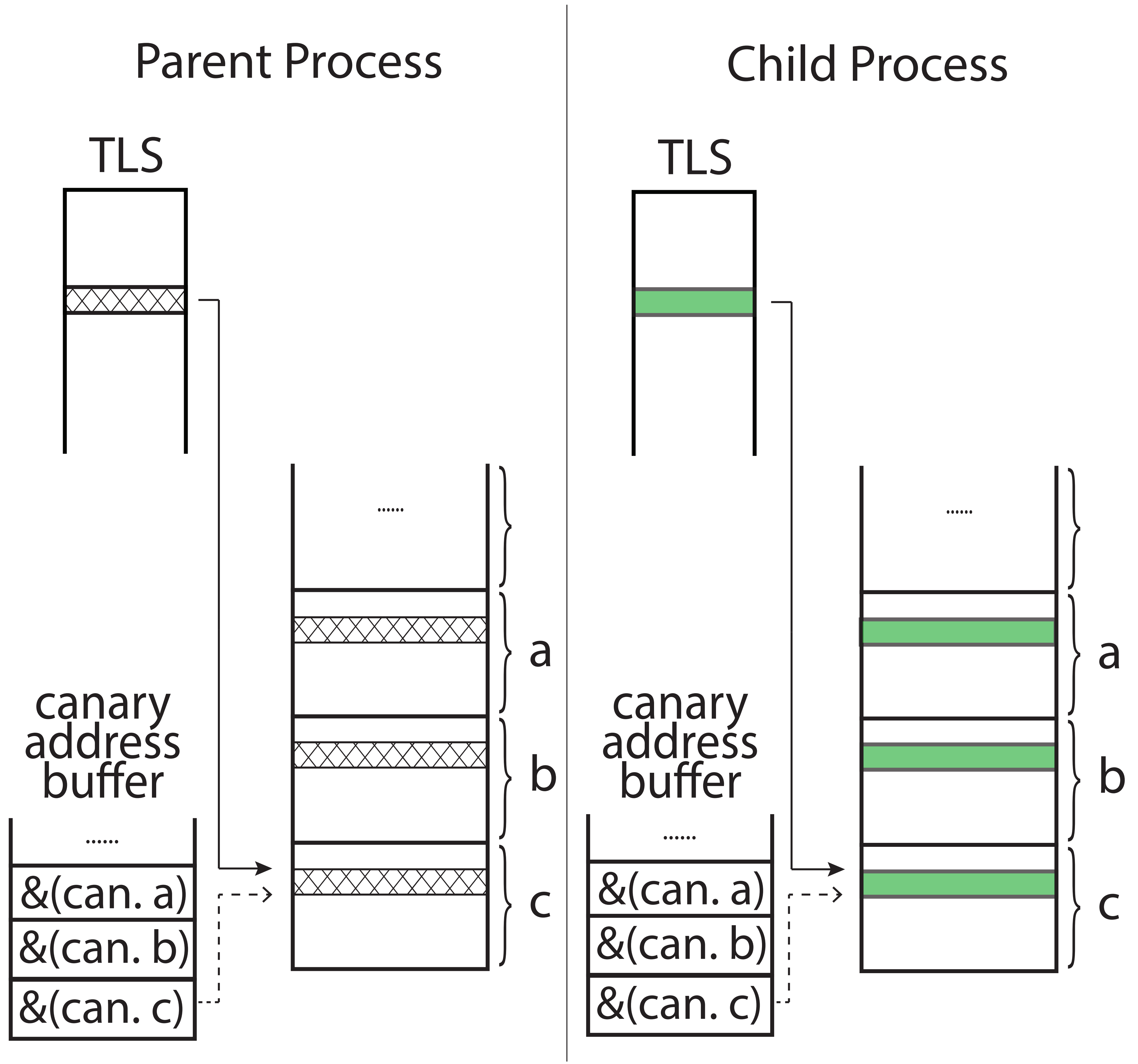


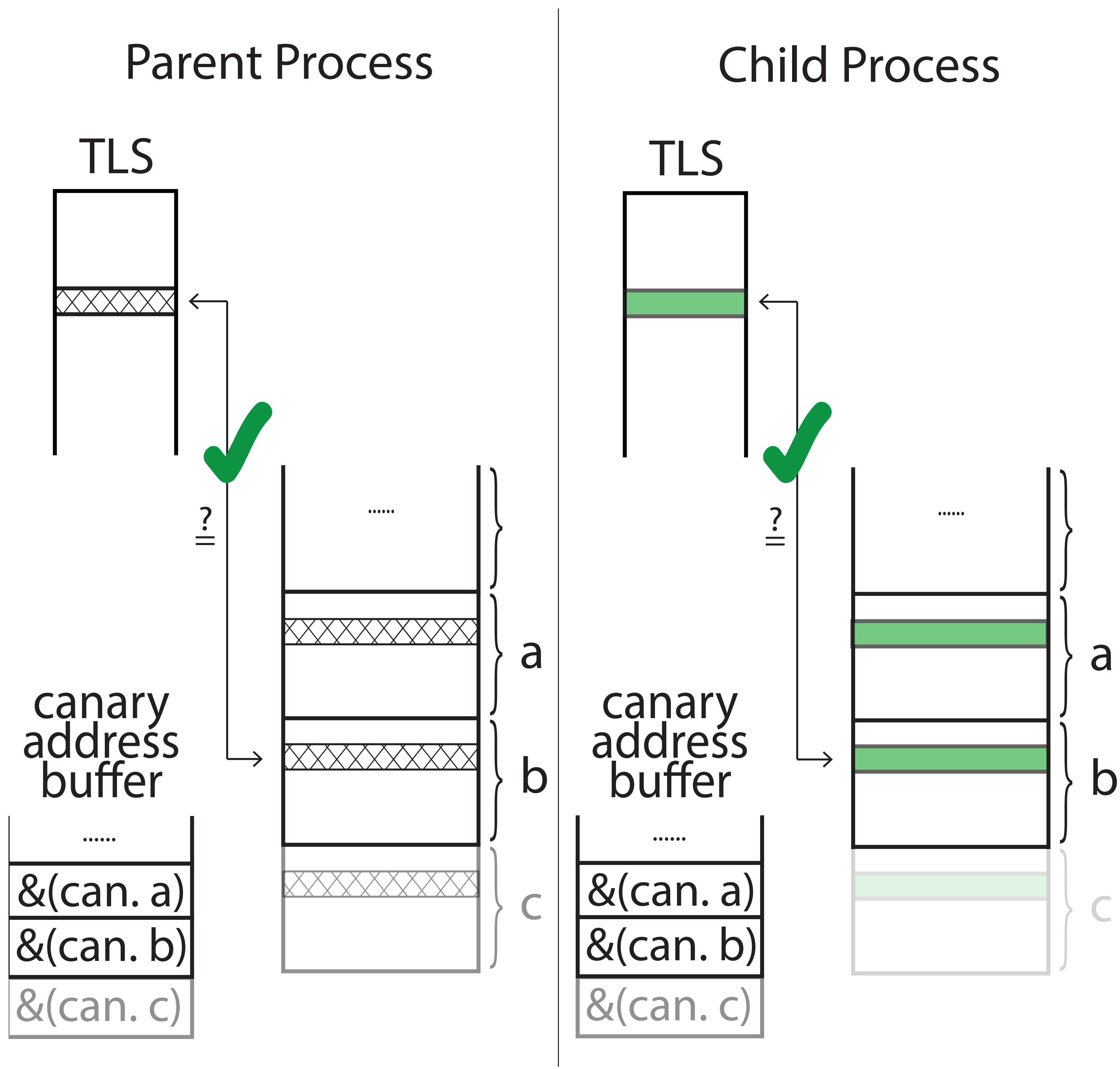
Parent Process
Child Process
TLS
TLS
......
......
?
=
?
=
a
b
c
a
b
c


canary
push
canary
reference
?
=
canary
check
TLS
canary
......
......
previous
frames
canary
address
buffer
a
......
&(canary a)
&(canary b)
b


Parent Process
TLS
......
a
b
canary
address
buffer
......
&(can. a)
&(can. b)


Child Process
TLS
......
a
b
canary
address
buffer
......
&(can. a)
&(can. b)


Parent Process
Child Process
TLS
TLS
......
a
b
c
canary address buffer
......
&(can. a)
&(can. b)
&(can. c)
......
a
b
c
canary address buffer
......
&(can. a)
&(can. b)
&(can. c)

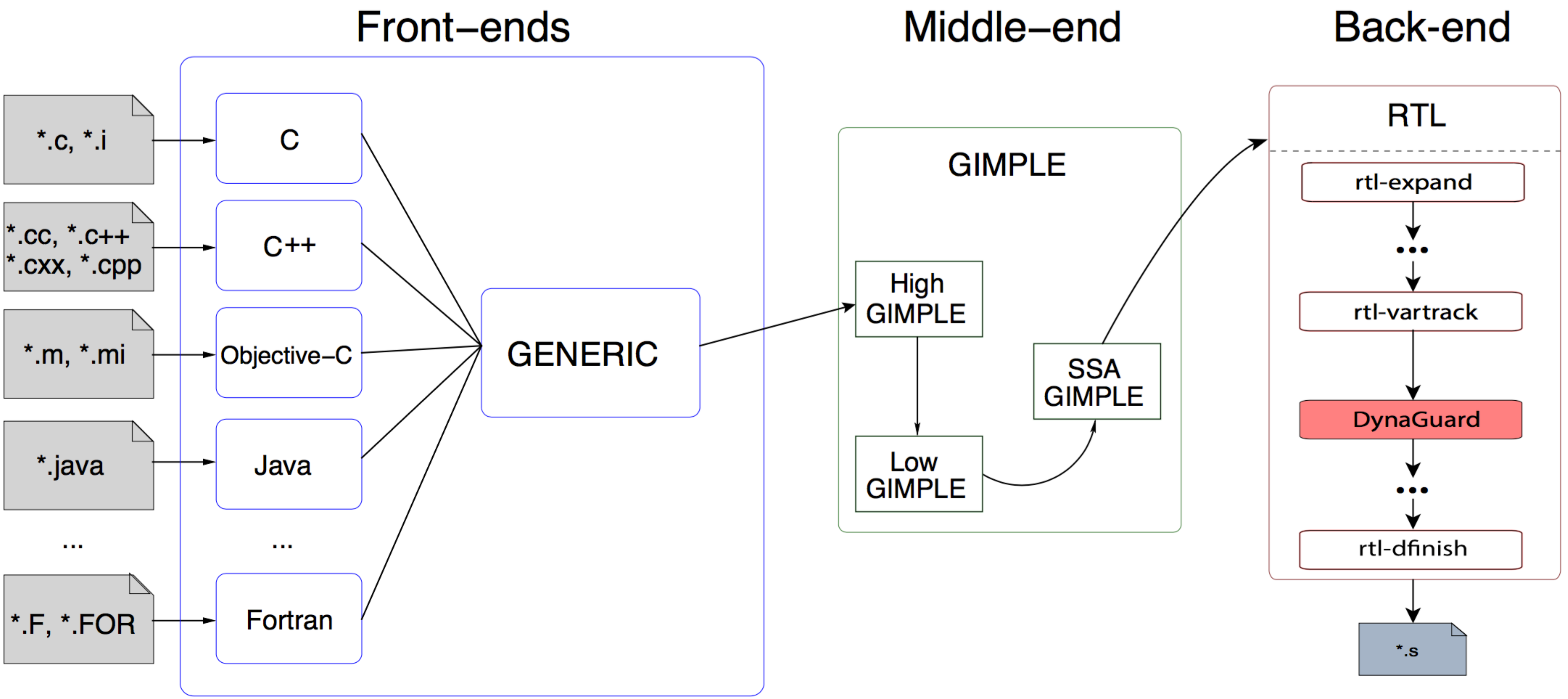


Parent Process
Child Process
TLS
TLS
......
?
=
a
b
c
canary
address
buffer
......
&(can. a)
&(can. b)
&(can. c)
......
?
=
a
b
c
canary
address
buffer
......
&(can. a)
&(can. b)
&(can. c)

Implementation

Two flavors: Compiler-based and DBI-based

Implementation: Compiler-based Version



Implementation: Compiler-based Version

- Two components:
  - GCC plugin
  - Runtime library
  - Total of ~1250 LOC

- Maintain two canaries at runtime:
  - DynaGuard-compiled code uses DynaGuard canaries
  - legacy code/libraries use the `glibc` canaries

Implementation: Compiler-based Version

- Both canaries have same entropy but are stored in different TLS offsets
- GCC plugin replaces the `glibc` canaries with the DynaGuard canaries
- DynaGuard's runtime library:
  - allocates Canary Address Buffer (CAB) in the heap for each thread, before it starts executing and deallocates it when terminating
  - performs CAB bookkeeping
  - updates all canaries in the child process's stack, as well as its TLS upon a fork()

Compiler-based Version: DynaGuard GCC Plugin

- Reserve 4 out of 8 `__padding` entries of the `tcbhead_t` struct in the TLS.
  Reserved TLS offsets range from 0x2a0 to 0x2b8:
  - CAB address stored at %fs:0x2a0
  - CAB current index: %fs:0x2a8
  - CAB size: %fs:0x2b0
  - DynaGuard canary: %fs:0x2b8

- Insert code to push/pop canary addresses in CAB upon a canary push/pop

Compiler-based Version: DynaGuard GCC Plugin

Original

```
;function prologue
push    %rbp
mov     %rsp,%rbp
sub     $0x40,%rsp
;canary stack placement
mov     %fs:0x28,%rax
mov     %rax,-0x8(%rbp)
xor     %eax,%eax

        ...

;canary check
mov     -0x8(%rbp),%rcx
xor     %fs:0x28,%rcx
je      <exit>
callq   <__stack_chk_fail@plt>
```

DynaGuard

```
push    %rbp
mov     %rsp,%rbp
sub     $0x40,%rsp
push    %r14                        (1)
push    %r15
lea     -0x8(%rbp),%rax             (2)
mov     %fs:0x2a0,%r14              (3)
mov     %fs:0x2a8,%r15              (4)
mov     %rax,(%r14,%r15,8)          (5)
incq    %fs:0x2a8                   (6)
pop     %r15                        (7)
pop     %r14
mov     %fs:0x2b8,%rax              (8)
mov     %rax,-0x8(%rbp)
xor     %eax,%eax
        ...

decq    %fs:0x2a8                   (9)
mov     -0x8(%rbp),%rcx
xor     %fs:0x2b8,%rcx              (10)
je      <exit>
callq   <__stack_chk_fail@plt>
```

Compiler-based Version: DynaGuard Runtime Library

- PIC module loaded via **`LD_PRELOAD`**
- Invoked only for CAB setup and resize operations, as well as for canary updates.
- All push/pop operations of canary addresses are implemented by the GCC plugin

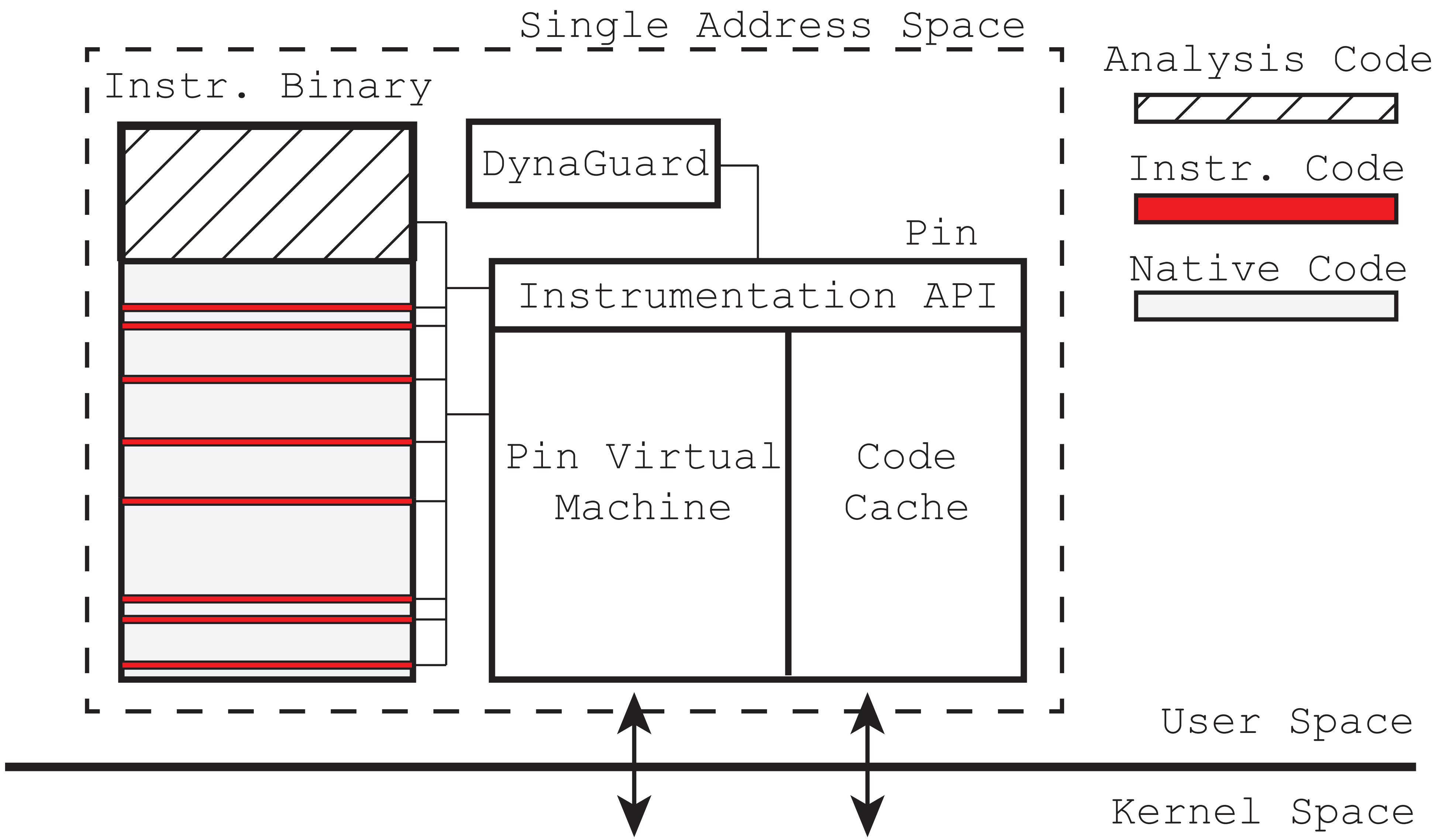
Compiler-based Version: DynaGuard Runtime Library

- Constructor routine allocates CAB in main thread
- Hooks:
  - `pthread_create` to setup the entries in TLS before `start_routine` starts executing
  - the `fork()` system call and updates all canaries in the child process's stack (before the child commences execution)
  - **stack unwinding routines** and updates the CAB accordingly
- Write-protects the last page of CAB, registers a `SIGSEGV` handler, and hooks `signal` and `sigaction`
  - If signal due to a full CAB, resize accordingly and resume execution
  - Else, invoke the original signal handler and let the application handle the signal

Implementation: DBI-based Version

Implemented using Intel's Pin DBI framework

- No source code needed
- Same design as previously except now execution occurs under Pin



Implementation: DBI-based Version

- Monitor all canary push and pop operations
- Update all canaries in the child process accordingly upon a **`fork`**
- No need for complex tracking of stack unwinding: simply track modifications of the stack pointer
- Maintain a per-thread CAB buffer, eliminating the overhead of using the Pin built-in **`trace buffer`**

Sample Function Prologue

```
push    rbp
mov     rsp,%rbp
sub     $0x40,rsp
mov     fs:0x28,%rax   (1)
mov     rax,-0x8(%rbp) (2)
```

Instrumentation Pseudocode

```
if((instruction has segment prefix)        &&
   (prefix is one of fs/gs)                &&
   (offset from fs/gs is 0x28/0x14)        &&
   (instr. is a 'mov' from mem to reg)     &&
   (next instr. is a `mov' from reg to mem)&&
   (dest. operand(register) of current instr.
    is the source operand of next instr.)) {
      insert_analysis_call(
            before_next_instr,
            push_canary(thread_context,
            canary_address))}
```

Evaluation

Effectiveness:

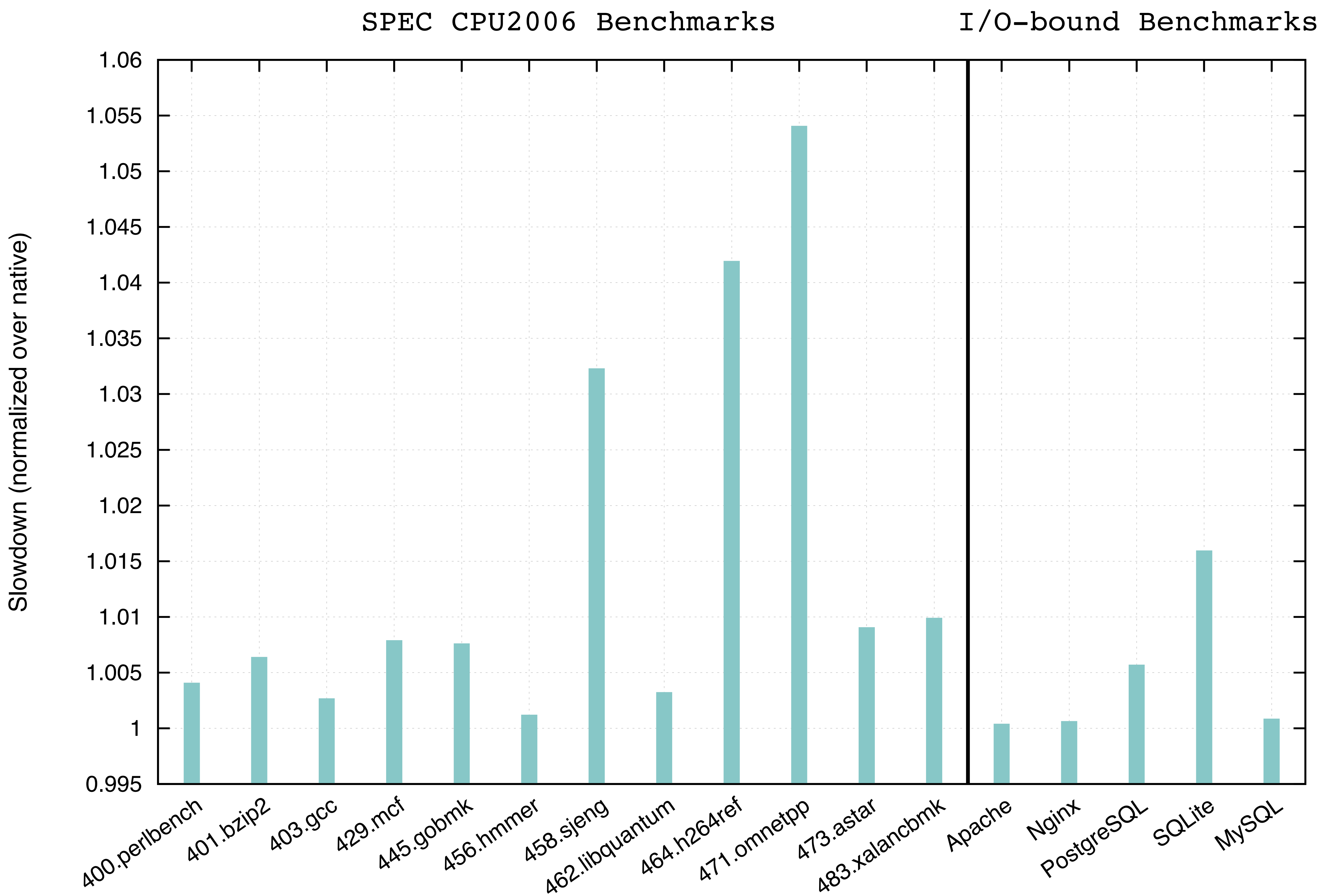
- Successfully defends against BROP and Nginx public exploits without breaking correctness

Performance:

- SPEC CPU 2006 INT benchmarks
- Popular Server Applications: Apache, Nginx, PostgreSQL, MySQL, SQLite
- Phoronix default profile for all server applications except MySQL (for which we used SysBench)
- Average overhead 1.2% in GCC version, 2.92% on top of PIN in DBI version

SPEC CPU2006: **1.5%**

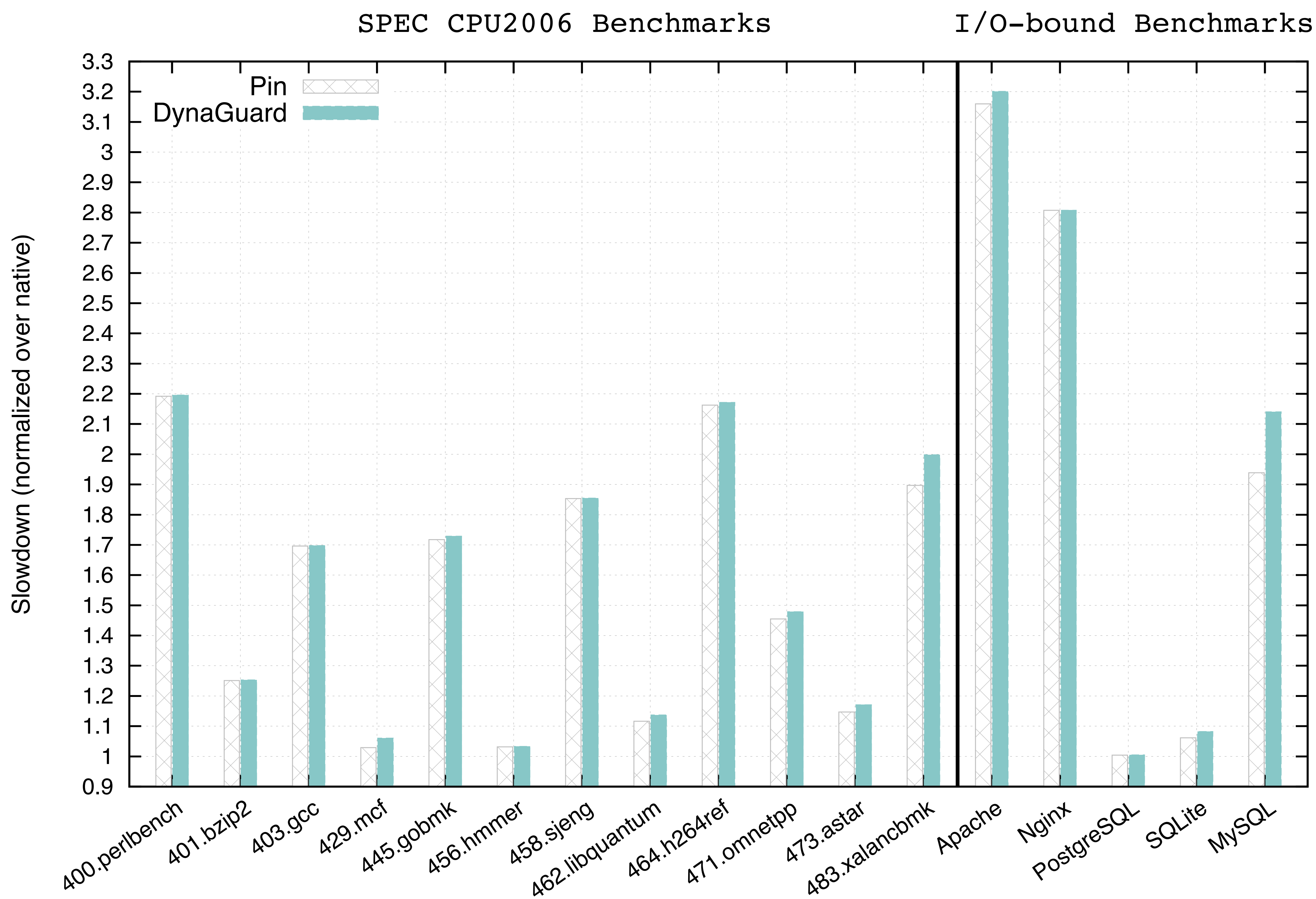
Server applications (Phoronix and SysBench): **0.46%**



SPEC CPU2006: **3.2% - 2.19x (avg 1.56x)**

PostgreSQL : **0.4%** - SQLite : **8.19%** - MySQL: **214%**- Apache: **3.2x** - Nginx: **2.8x**

Average CPU overhead **170.66%, 2.92%** atop PIN



Summary

- DynaGuard protects canary-based defenses against byte-by-byte brute forcing of the canary cookie

- Supports applications for which source code is available as well as binary-only programs
  - Offers a lightweight solution for the more general problem of memory duplication with respect to reduced entropy for security-sensitive applications (e.g., PRNGs of OpenSSL and LibreSSL)

- Has minimal incremental overhead over the respective underlying protection (e.g., GCC's SSP & Pin's native DBI respectively)

- Source code is available at https://github.com/nettrino/dynaguard