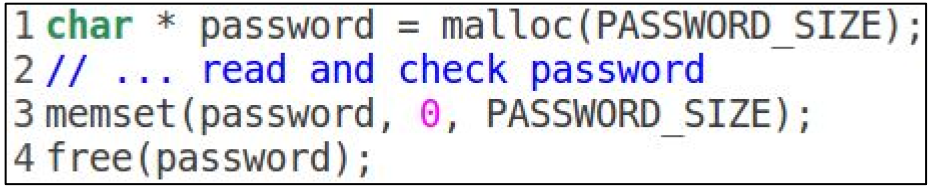
Dead Store Elimination (Still) Considered Harmful

Zhaomo Yang[1], Brian Johannesmeyer[1], Anders Trier Olesen[2], Sorin Lerner[1] and Kirill Levchenko[1]

[1] *UC San Diego*
[2] *Aalborg University*

Motivation

- Concerns over memory disclosure vulnerabilities in C and C++ programs have led developers to explicitly scrub sensitive data from memory.



```c
char * password = malloc(PASSWORD_SIZE);
// ... read and check password
memset(password, 0, PASSWORD_SIZE);
free(password);
```

- However, ***Dead Store Elimination*** **(DSE)** removes stores that have no effect on the program result.
- Security-conscious developers have been aware of this phenomenon and have devised ways to circumvent it.

Outline

Goal. Understanding the current state of the dead store elimination problem and developers' attempts to circumvent it.

> **Existing Techniques.** A survey of existing techniques used to scrub memory found in open source security projects.
>
> **Case Studies.** An analysis of eleven projects to understand the use of memory scrubbing in real world programs.
>
> **Our solutions.** A single best-of-breed scrubbing function and a scrubbing-safe dead store elimination optimization pass.

Existing Techniques

For each technique, I will first describe how it is intended to work, its availability on different platforms, and its effectiveness.

We rate the effectiveness of a technique on a three-level scale:

- **Effective.** Guaranteed to work (barring flaws in implementation).
- **Effective in practice.** Works with all compilation options and on all the compilers we tested (GCC, Clang, and MSVC), but is not guaranteed in principle.
- **Flawed.** Fails in at least one configuration.

Existing Techniques: Platform-Supplied Functions

Platform-supplied scrubbing functions that guarantee the desired behavior:

- SecureZeroMemory (on Windows)
- explicit_bzero (on OpenBSD and NetBSD and in glibc)

- ***Used in***: Kerberos's zap, Libsodium's sodium_memzero, Tor's memwipe, Libsodium's sodium_memzero, Tor's memwipe, OpenSSH's explicit_bzero.
- ***Availability***: only on certain platforms or in certain versions of a specific library
- ***Effectiveness***: 🟩 effective

Existing Techniques: Platform-Supplied Functions

The latest C standard (ISO/IEC 9899-2011) introduced memset_s, declared as

```
errno_t memset_s( void *dest, rsize_t destsz, int ch, rsize_t count )
```

memset_s is considered as a secure version of memset because

- It does some runtime checking of its parameters, and
- **Calls to it can never be optimized out.**

Existing Techniques: Platform-Supplied Functions

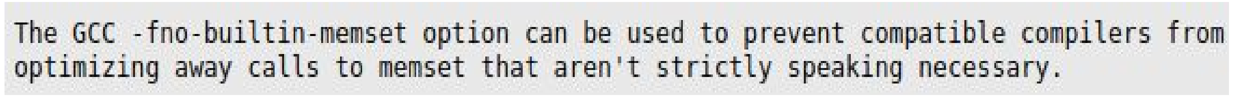
Possible reasons for the absence of implementation of `memset_s`:

- `memset_s` is part of the ***optional*** Annex K
- In addition, C11 treats all the functions in the Annex K as a unit. That is, if a C library wants to implement memset_s in a standard-conforming fashion, it has to implement ***all*** of the functions defined in this annex.
- Library developers also argued that **some functions are poorly designed** [1].

[1] https://sourceware.org/ml/libc-alpha/2014-12/msg00506.html

Existing Techniques: Using -fno-builtin-memset

In a thread that requests a glibc implementation of memset_s, a glibc developer suggested the requester to use the **-fno-builtin-memset** option instead [1].



The GCC -fno-builtin-memset option can be used to prevent compatible compilers from optimizing away calls to memset that aren't strictly speaking necessary.

[1] https://sourceware.org/bugzilla/show_bug.cgi?id=17879

Existing Techniques: Using -fno-builtin-memset

***How developers expect it to work*:**

- To improve performance, compilers replaces calls to memset with its built-in equivalent.
- Compilers only knows the semantics of the built-in memset, not the memset from the C standard library.

***Why this technique is not guaranteed to work (in theory)*:**

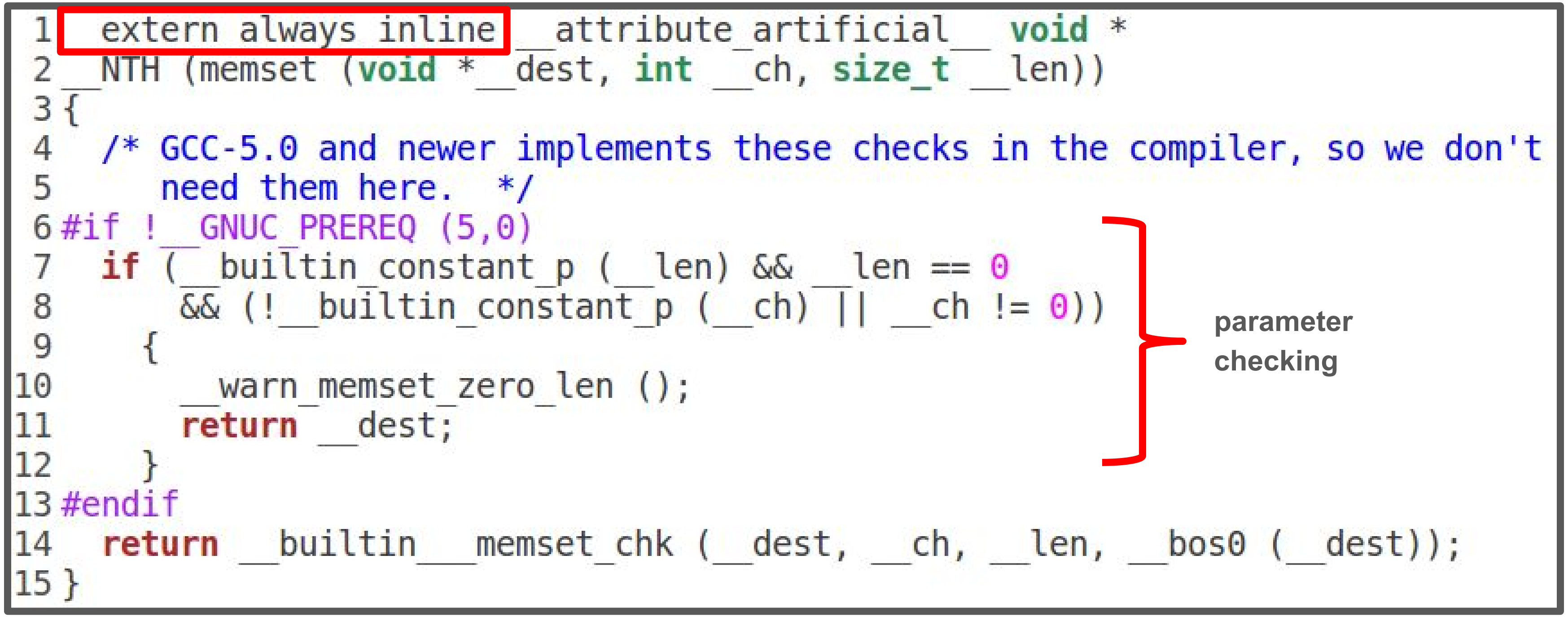
- Disabling the built-in memset does not prevent the compiler from knowing the semantics of the C library memset, which is specified by the C standard.

Existing Techniques: Using -fno-builtin-memset

Why this technique is not guaranteed to work (in practice):

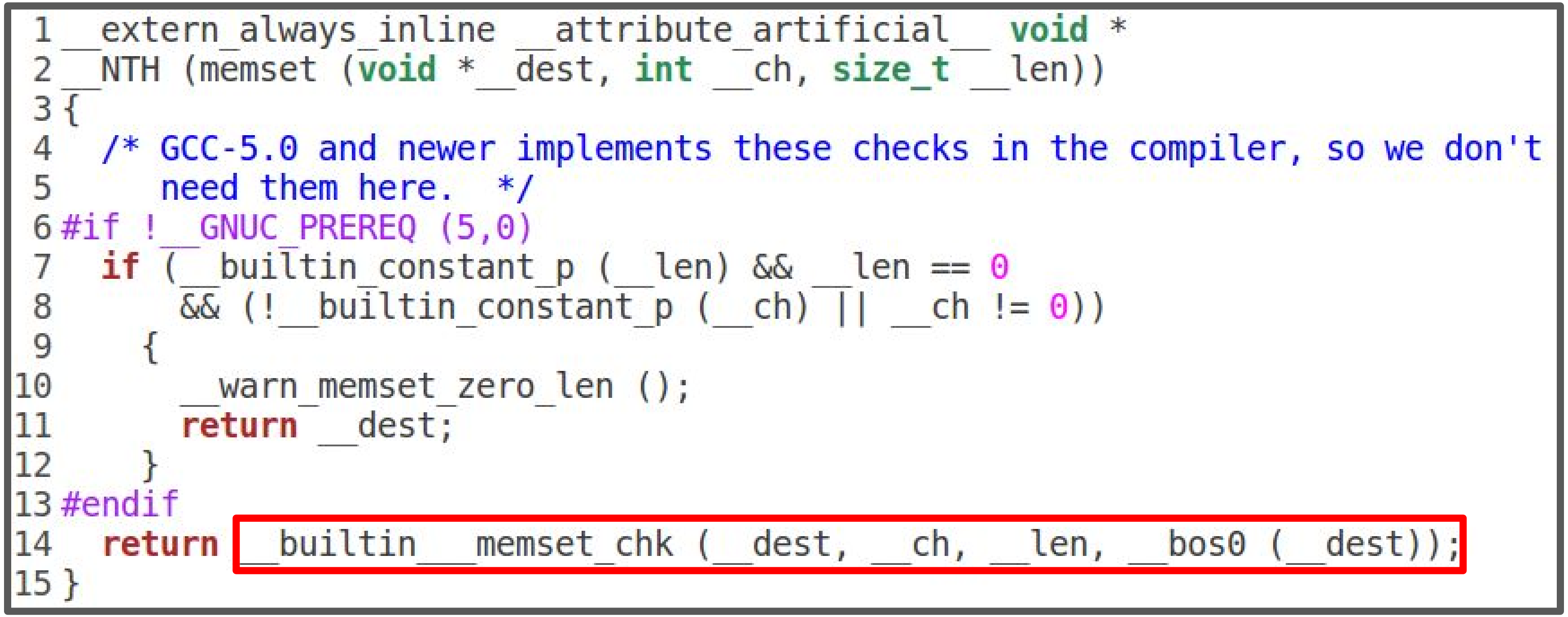
Staring from **glibc >= 2.3.4**, when **the optimization level > O0** and **the macro _FORTIFY_SOURCE > 0**, the fortified version of memset is enabled, ***regardless of whether -fno-builtin-memset is used or not***.

Existing Techniques: Using -fno-builtin-memset



```
__extern_always_inline __attribute_artificial__ void *
__NTH (memset (void *__dest, int __ch, size_t __len))
{
  /* GCC-5.0 and newer implements these checks in the compiler, so we don't
     need them here.  */
#if !__GNUC_PREREQ (5,0)
  if (__builtin_constant_p (__len) && __len == 0
      && (!__builtin_constant_p (__ch) || __ch != 0))
    {
      __warn_memset_zero_len ();
      return __dest;
    }
#endif
  return __builtin___memset_chk (__dest, __ch, __len, __bos0 (__dest));
}
```

Existing Techniques: Using -fno-builtin-memset



```c
__extern_always_inline __attribute_artificial__ void *
__NTH (memset (void *__dest, int __ch, size_t __len))
{
  /* GCC-5.0 and newer implements these checks in the compiler, so we don't
     need them here.  */
#if !__GNUC_PREREQ (5,0)
  if (__builtin_constant_p (__len) && __len == 0
      && (!__builtin_constant_p (__ch) || __ch != 0))
    {
      __warn_memset_zero_len ();
      return __dest;
    }
#endif
  return __builtin___memset_chk (__dest, __ch, __len, __bos0 (__dest));
}
```

Existing Techniques: Using -fno-builtin-memset

In summary:

This technique is flawed not only in theory but also in practice.

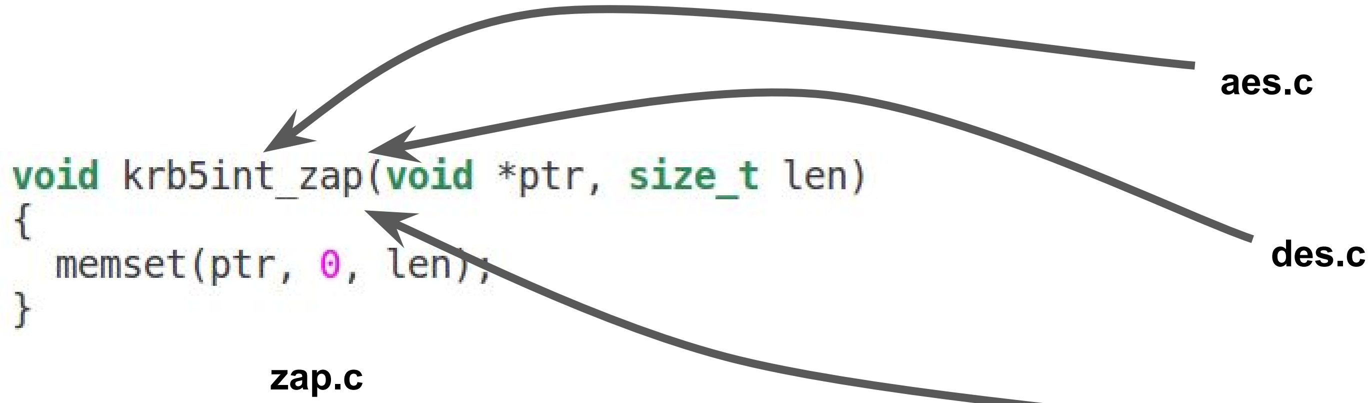
- ***Availability***: Widely available
- ***Effectiveness***: **flawed**

Existing Techniques

Hiding Semantics. If the compiler doesn’t recognize that an operation is clearing memory, it will not remove it.

Hiding Semantics: Separate Compilation

The simplest way to hide the semantics from the compiler is to **implement the scrubbing operation in a separate compilation unit**.



```
void krb5int_zap(void *ptr, size_t len)
{
  memset(ptr, 0, len);
}
```

zap.c from Kerberos

Hiding Semantics: Separate Compilation

***How developers expect it to work*:**

Defining the scrubbing function in a separate compilation unit will prevent the compiler from inlining and understanding it in the calling function.

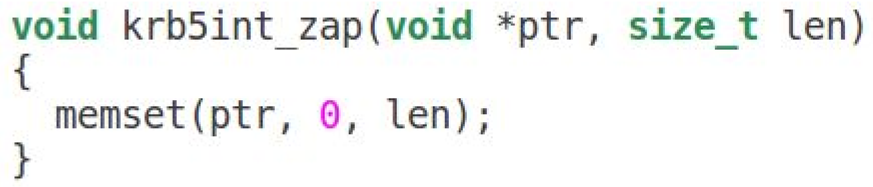
When it is not guaranteed to work:

When Link-Time Optimization (LTO) is enabled, this technique will not work.

Link-Time Optimization (LTO)

Link-Time Optimization (LTO) can merge all compilation units into one and then perform regular optimizations (including DSE) on the single compilation unit.

With LTO enabled, such a scrubbing function can be inlined in a calling function, and the call to memset will be subject to DSE.



```
void krb5int_zap(void *ptr, size_t len)
{
  memset(ptr, 0, len);
}
```

Hiding Semantics: Separate Compilation

How developers expect it to work:

Defining the scrubbing function in a separate compilation unit will prevent the compiler from inlining and understanding it in the calling function.

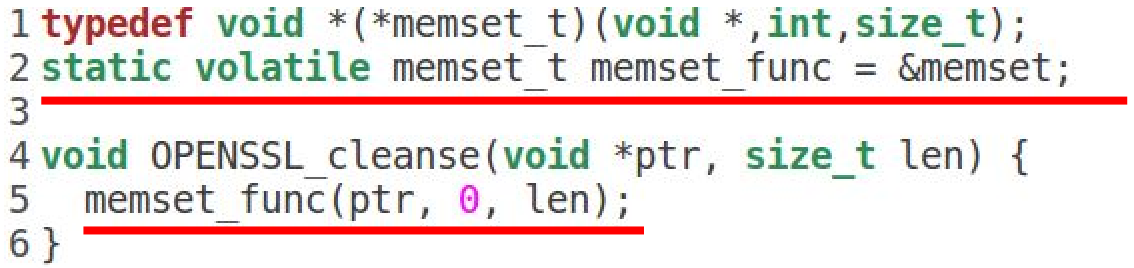
Why it is not guaranteed to work:

When Link-Time Optimization (LTO) is enabled, this technique will not work.

- ***Used in***: Kerberos' `zap`
- ***Availability***: Universal
- ***Effectiveness***: **flawed**

Existing Techniques: Volatile Function Pointer

OPENSSL_cleanse (since OpenSSL 1.0.2) is one of the implementations based on this idea.



```
typedef void *(*memset_t)(void *,int,size_t);
static volatile memset_t memset_func = &memset;

void OPENSSL_cleanse(void *ptr, size_t len) {
  memset_func(ptr, 0, len);
}
```

Existing Techniques: Volatile Function Pointer

***How developers expect it to work*:**

- The call to memset via a volatile function pointer is a volatile access, which the compiler cannot optimize out.

Why it is not guaranteed to work:

- This behavior is **not** guaranteed by the C standard.

Existing Techniques: Volatile Function Pointer

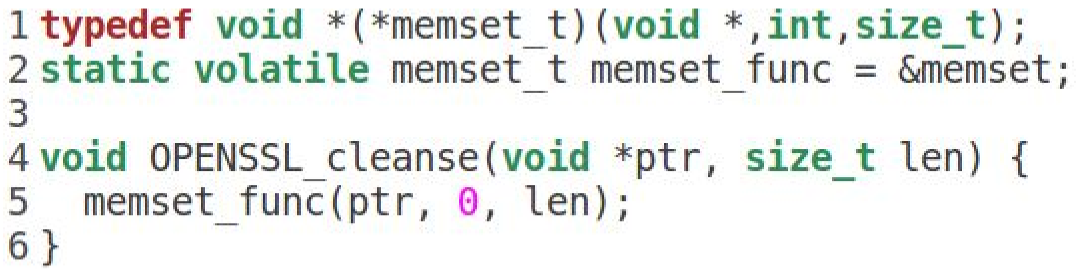
The C11 standard defines an object of volatile-qualified type as follows:

> “An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously. What constitutes an access to an object that has volatile-qualified type is implementation-defined.”

In summary,

- A compliant compiler cannot optimize out any volatile access.
- A compliant compiler is free to decide what constitutes a volatile access.

Existing Techniques: Volatile Function Pointer



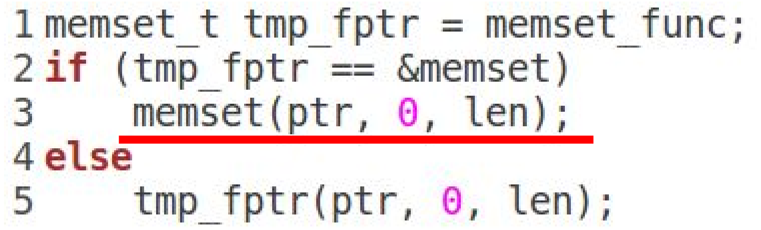
```
1 typedef void *(*memset_t)(void *,int,size_t);
2 static volatile memset_t memset_func = &memset;
3
4 void OPENSSL_cleanse(void *ptr, size_t len) {
5   memset_func(ptr, 0, len);
6 }
```

Line 5 consists of two steps:

- Reading the volatile pointer memset_func
- Calling the function pointed by memset_func

Existing Techniques: Volatile Function Pointer

In theory, such a compiler may inline each call to OPENSSL_cleanse as:



```c
memset_t tmp_fptr = memset_func;
if (tmp_fptr == &memset)
    memset(ptr, 0, len);
else
    tmp_fptr(ptr, 0, len);
```

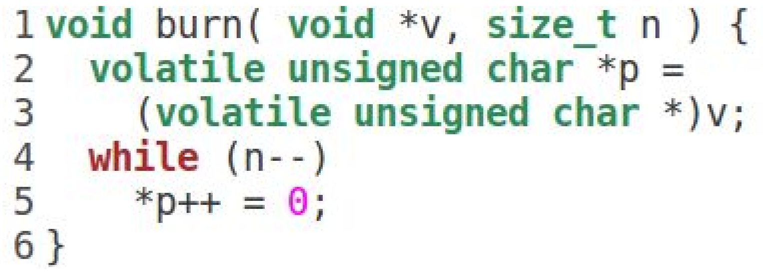
- ***Used in***: OpenSSL 1.0.2's OPENSSL_cleanse (also used in Tor and Bitcoin); OpenSSH's explicit_bzero, quarkslab's memset_s
- ***Availability***: Universal
- ***Effectiveness***: **effective in practice**

Existing Techniques

Forcing Memory Writes. Attempts to force the compiler to include the scrubbing operation without hiding its nature.

Existing Techniques: Pointer To Volatile Char

A popular way to force the compiler to perform a store is using a **pointer to volatile char**.



```c
void burn( void *v, size_t n ) {
  volatile unsigned char *p =
    (volatile unsigned char *)v;
  while (n--)
    *p++ = 0;
}
```

Existing Techniques: Pointer To Volatile Char

***How developers expect it to work*:**

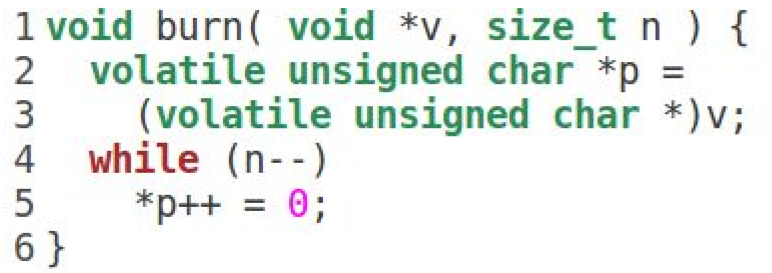
- Memory writes via pointer to volatile char **p** are volatile access, which the compiler cannot optimize out.

Why it is not guaranteed to work:

- This behavior is **not** guaranteed by the C standard.

Is accessing a non-volatile object via a pointer to volatile char a volatile access?

Existing Techniques: Pointer To Volatile Char



```c
void burn( void *v, size_t n ) {
  volatile unsigned char *p =
    (volatile unsigned char *)v;
  while (n--)
    *p++ = 0;
}
```

- *Used in*: sodium_memzero from Libsodium, insecure_memzero from Tarsnap, wipememory from Libgcrypt, SecureWipeBuffer from Crypto++, burn from Cryptography Coding Standard, ForceZero from wolfSSL, sudo_memset_s from sudo, and CERT’s C99-compliant solution.
- *Availability*: Universal
- *Effectiveness*: **effective in practice**

Existing Technique: Using memory barrier

GCC supports a memory barrier expressed using an inline assembly statement.

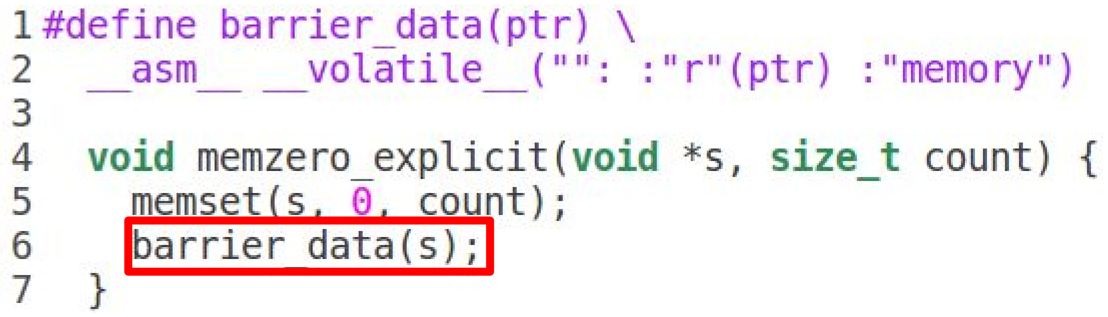
```
__asm__ __volatile__("":::"memory")
```

clobber argument

According to GCC’s documentation, the clobber argument "memory" tells the compiler that the inline assembly statement may read or write memory that is not specified in the input or output arguments.

Forcing Memory Writes: Using memory barrier

memzero_explicit from Linux uses memory barrier to force writes

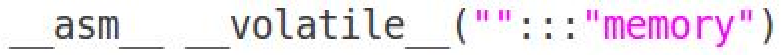


```
#define barrier_data(ptr) \
  __asm__ __volatile__("": :"r"(ptr) :"memory")

  void memzero_explicit(void *s, size_t count) {
    memset(s, 0, count);
    barrier_data(s);
  }
```

- *Used in*: zap from Kerberos, memzero_explicit from Linux.
- *Availability*: GCC and Clang.
- *Effectiveness*: **effective**

How difficult to create a reliable scrubbing function



```
__asm__ __volatile__("":::"memory")
```

GCC’s documentation indicates that the following inline assembly should work as a memory barrier. In practice, it does work with GCC.

Since Clang also supports barriers with the same syntax, and in general it mimics GCC’s behaviors, one would expect that the barrier above would also work with Clang.

How difficult to create a reliable scrubbing function

However, it does **not** work with Clang.

```
__asm__ __volatile__("":::"memory")
```

Unreliable with CLang

A more reliable and portable memory barrier is shown below (which is also used used in memzero_explicit):

```
__asm__ __volatile__("": :"r"(ptr) :"memory")
```

Reliable

Outline

Goal. Understanding the current state of the dead store elimination problem and developers’ attempts to circumvent it.

Existing Techniques. A survey of existing techniques used to scrub memory found in open source security projects.

Case Studies. An analysis of eleven security projects to determine whether a memory scrubbing function is available, effective, and used consistently.

Our solutions. A single best-of-breed scrubbing function that combines the most reliable techniques found in use today, and a scrubbing-safe dead store elimination optimization pass.

Case Studies

To understand the use of memory scrubbing in practice, we examined the eleven open source libraries and applications: **NSS**, **OpenVPN**, **Kerberos**, **Libsodium**, **Tarsnap**, **Libgcrypt**, **Crypto++**, **Tor**, **Bitcoin**, **OpenSSH** and **OpenSSL**.

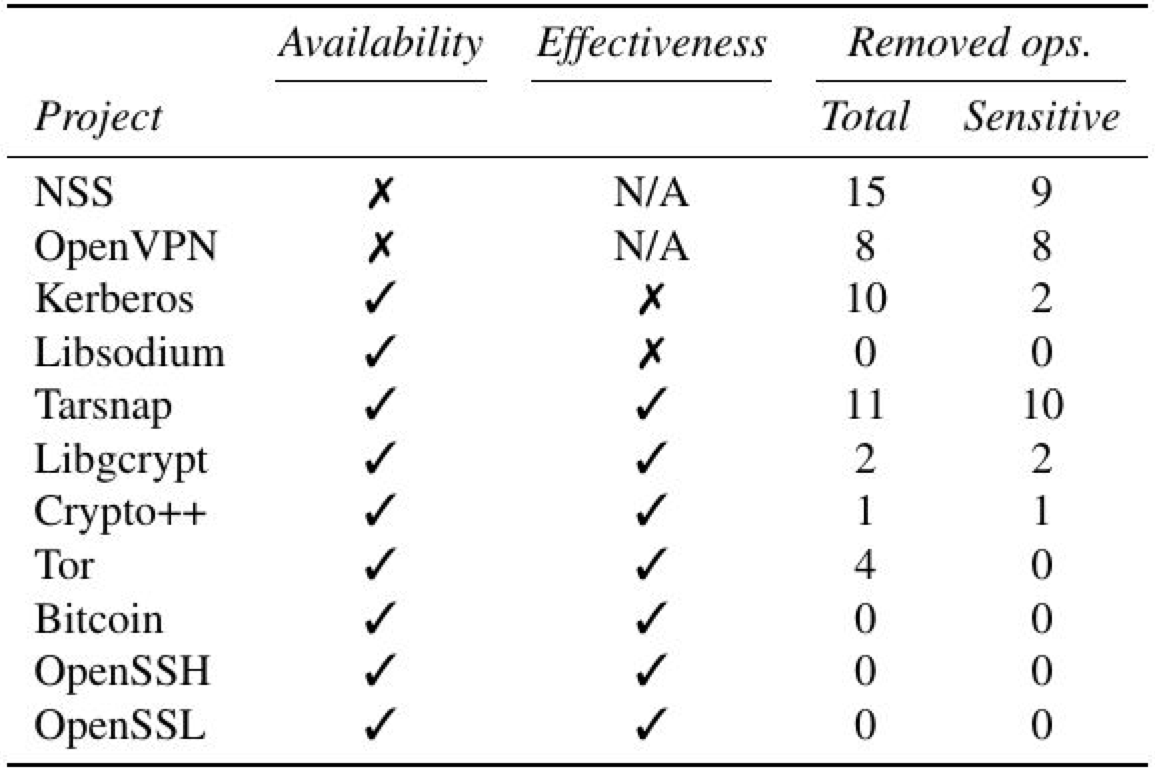
For each project, we set out to determine whether a memory scrubbing function is **available**, **effective** and **used consistently**.

Case Studies: Methodology

Our methodology consists of two parts:

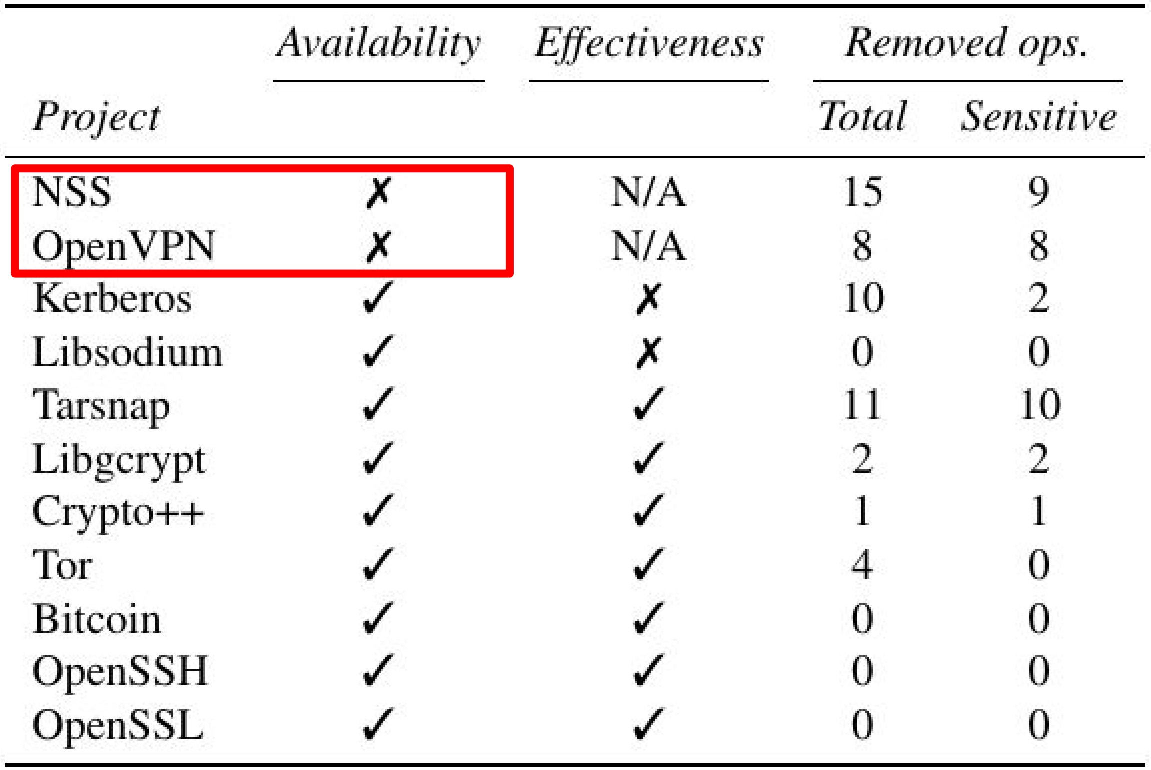
- we manually analyzed each project to determine whether a memory scrubbing function is available and whether it is effective.
- we instrumented the Clang 3.9 compiler to report instances of dead store elimination where a write is eliminated because the memory location is not used afterwards. For each project, we used this compiler to determine whether the memory scrubbing function was effective and used consistently.

Case Studies: Results



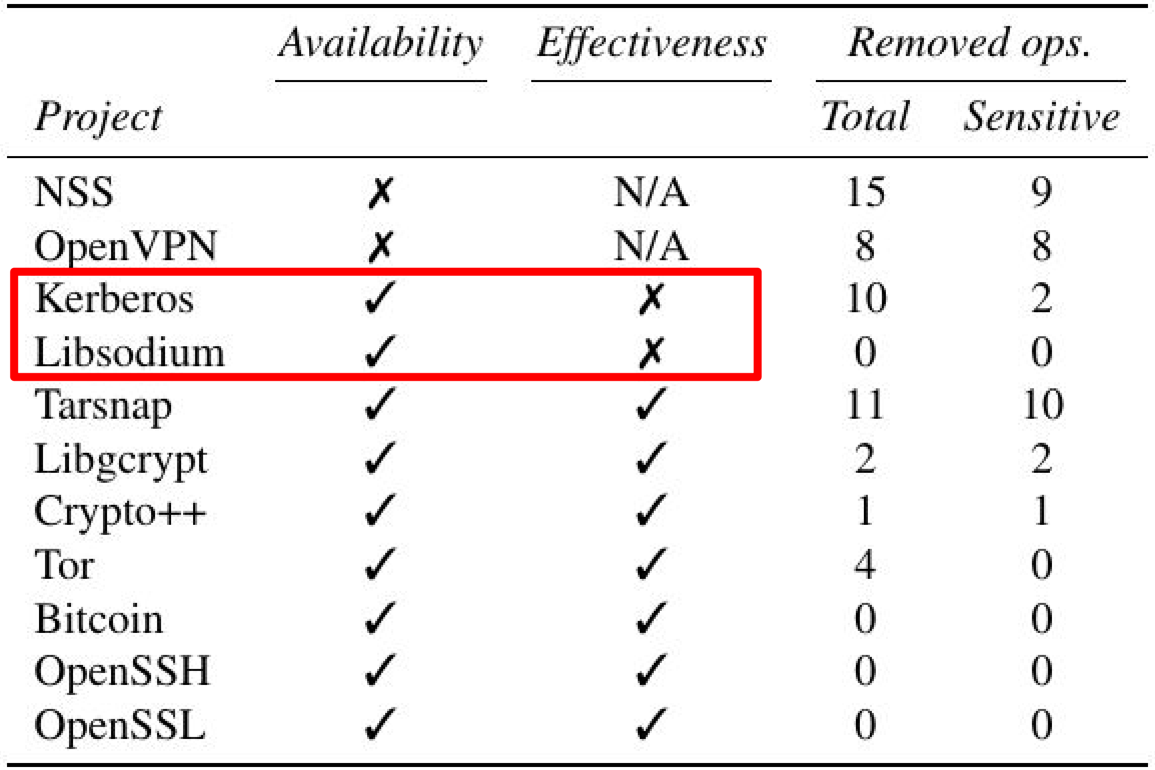
Project	*Availability*	*Effectiveness*	*Removed ops.*	
			Total	*Sensitive*
NSS	✗	N/A	15	9
OpenVPN	✗	N/A	8	8
Kerberos	✓	✗	10	2
Libsodium	✓	✗	0	0
Tarsnap	✓	✓	11	10
Libgcrypt	✓	✓	2	2
Crypto++	✓	✓	1	1
Tor	✓	✓	4	0
Bitcoin	✓	✓	0	0
OpenSSH	✓	✓	0	0
OpenSSL	✓	✓	0	0

Case Studies: Results



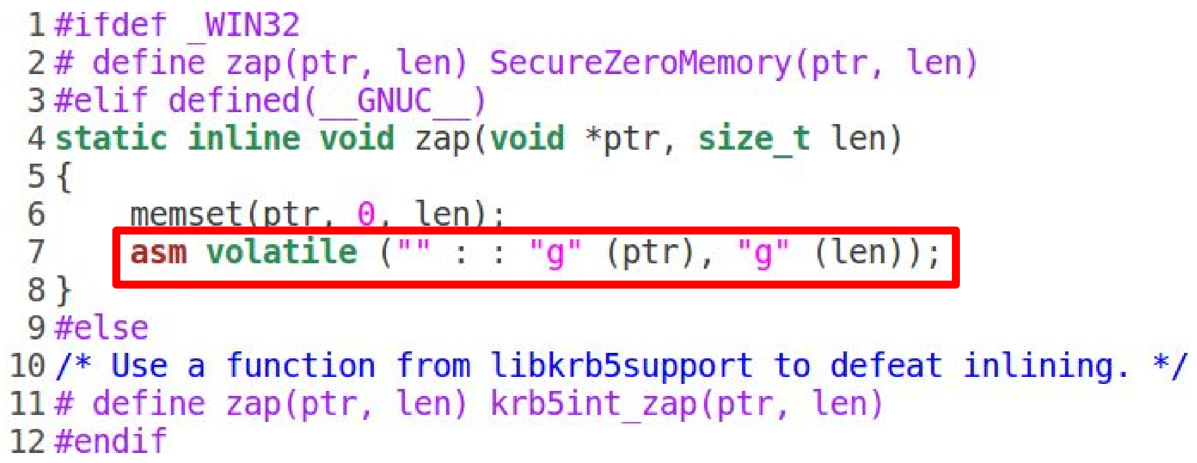
Project	*Availability*	*Effectiveness*	*Removed ops.*	
			Total	*Sensitive*
NSS	✗	N/A	15	9
OpenVPN	✗	N/A	8	8
Kerberos	✓	✗	10	2
Libsodium	✓	✗	0	0
Tarsnap	✓	✓	11	10
Libgcrypt	✓	✓	2	2
Crypto++	✓	✓	1	1
Tor	✓	✓	4	0
Bitcoin	✓	✓	0	0
OpenSSH	✓	✓	0	0
OpenSSL	✓	✓	0	0

Case Studies: Results



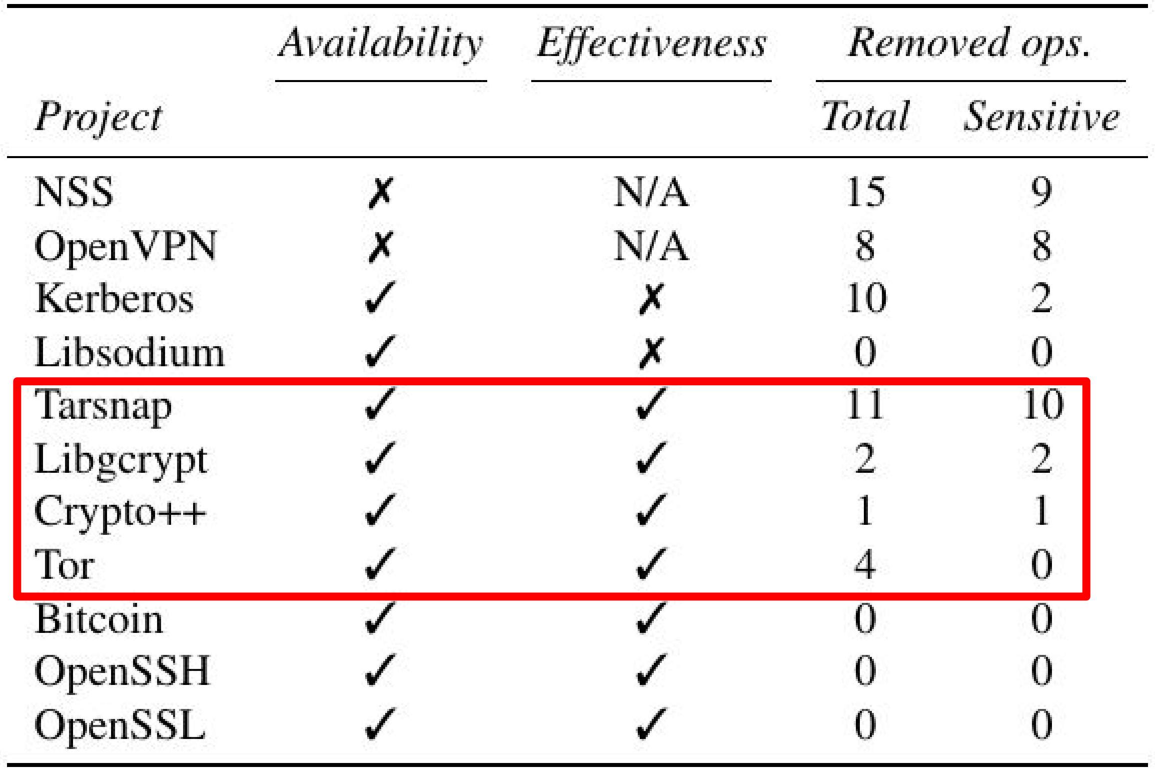
Project	*Availability*	*Effectiveness*	*Removed ops.*	
			Total	*Sensitive*
NSS	✗	N/A	15	9
OpenVPN	✗	N/A	8	8
Kerberos	✓	✗	10	2
Libsodium	✓	✗	0	0
Tarsnap	✓	✓	11	10
Libgcrypt	✓	✓	2	2
Crypto++	✓	✓	1	1
Tor	✓	✓	4	0
Bitcoin	✓	✓	0	0
OpenSSH	✓	✓	0	0
OpenSSL	✓	✓	0	0

zap from Kerbros



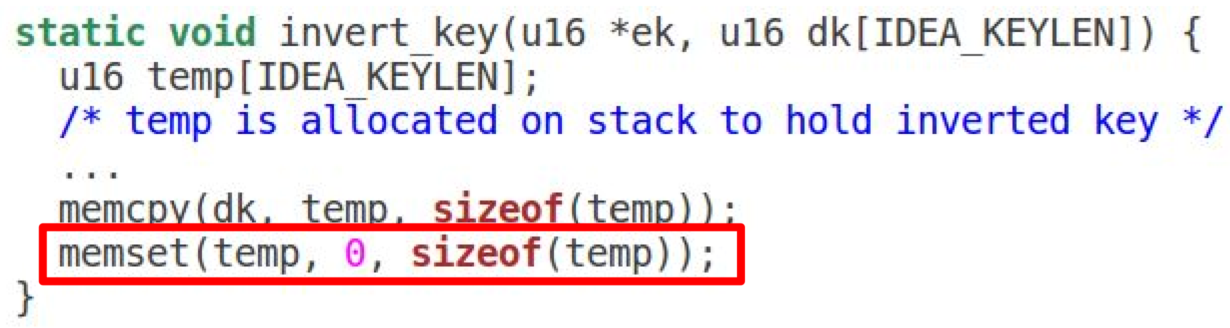
```c
#ifdef _WIN32
# define zap(ptr, len) SecureZeroMemory(ptr, len)
#elif defined(__GNUC__)
static inline void zap(void *ptr, size_t len)
{
    memset(ptr, 0, len);
    asm volatile ("" : : "g" (ptr), "g" (len));
}
#else
/* Use a function from libkrb5support to defeat inlining. */
# define zap(ptr, len) krb5int_zap(ptr, len)
#endif
```

Case Studies: Results



Project	*Availability*	*Effectiveness*	*Removed ops.*	
			Total	*Sensitive*
NSS	✗	N/A	15	9
OpenVPN	✗	N/A	8	8
Kerberos	✓	✗	10	2
Libsodium	✓	✗	0	0
Tarsnap	✓	✓	11	10
Libgcrypt	✓	✓	2	2
Crypto++	✓	✓	1	1
Tor	✓	✓	4	0
Bitcoin	✓	✓	0	0
OpenSSH	✓	✓	0	0
OpenSSL	✓	✓	0	0

invert_key from Libgcrypt



```
static void invert_key(u16 *ek, u16 dk[IDEA_KEYLEN]) {
  u16 temp[IDEA_KEYLEN];
  /* temp is allocated on stack to hold inverted key */
  ...
  memcpy(dk, temp, sizeof(temp));
  memset(temp, 0, sizeof(temp));
}
```

Case Studies: Discussion

Our case studies lead us to two observations.

- **There is no single accepted scrubbing function**. Each project mixes its own cocktail using existing scrubbing techniques, some of which are flawed or unreliable
- Even when a project has a reliable scrubbing function, **the developers may not use it consistently**.

Our Solutions

- **Library-based solution**. secure_memzero
- **Compiler-based solution**. Scrubbing-aware DSE

Our Solutions: Library-based solution

Library-based solution. secure_memzero

- This function combines the effective scrubbing techniques we found in a simple implementation.
- Developers can specify an order of preference in which an implementation will be chosen by defining macros.
- We have released our implementation into the public domain, and we plan to keep our implementation updated to ensure it remains effective as compilers evolve.

https://compsec.sysnet.ucsd.edu/secure_memzero.h

Our Solutions: Compiler-based solution

Compiler-based solution. Scrubbing-aware DSE

This DSE pass considers a memory write satisfying the following conditions as a scrubbing operation:

- The stored value is a constant,
- The number of bytes stored is a constant, and
- The store is subject to elimination because the variable is about to be out of scope without being read

Conclusion

- We surveyed the existing solutions to circumvent the dead store elimination problem.
- Our case studies show that real world programs still have unscrubbed sensitive data, due to incorrect implementation of scrubbing function as well as from developers simply forgetting to use the secure scrubbing function.
- To solve the problem, we developed a scrubbing-aware DSE pass that preserves all scrubbing operations and `secure_memzero`, a best-of-breed scrubbing function.

Questions