Dead Store Elimination (Still) Considered Harmful

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

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

Abstract

Dead store elimination is a widely used compiler optimization that reduces code size and improves performance. However, it can also remove seemingly useless memory writes that the programmer intended to clear sensitive data after its last use. Security-savvy developers have long been aware of this phenomenon and have devised ways to prevent the compiler from eliminating these data scrubbing operations.

In this paper, we survey the set of techniques found in the wild that are intended to prevent data-scrubbing operations from being removed during dead store elimination. We evaluated the effectiveness and availability of each technique and found that some fail to protect data-scrubbing writes. We also examined eleven open source security projects to determine whether their specific memory scrubbing function was effective and whether it was used consistently. We found four of the eleven projects using flawed scrubbing techniques that may fail to scrub sensitive data and an additional four projects not using their scrubbing function consistently. We address the problem of dead store elimination removing scrubbing operations with a compiler-based approach by adding a new option to an LLVM-based compiler that retains scrubbing operations. We also synthesized existing techniques to develop a best-of-breed scrubbing function and are making it available to developers.

1 Introduction

Concerns over memory disclosure vulnerabilities in C and C++ programs have long led security application developers to explicitly *scrub* sensitive data from memory. A typical case might look like the following:

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

The `memset` is intended to clear the sensitive `password` buffer after its last use so that a memory disclosure vulnerability could not reveal the password. Unfortunately, compilers perform an optimization—called *dead store elimination* (DSE)—that removes stores that have no effect on the program result, either because the stored value is overwritten or because it is never read again. In this case, because the buffer is passed to `free` after being cleared, the compiler determines that the memory scrubbing `memset` has no effect and eliminates it.

Removing buffer scrubbing code is an example of what D'Silva *et al.* [29] call a "correctness-security gap:" From the perspective of the C standard, removing the `memset` above is allowed because the contents of unreachable memory are not considered part of the semantics of the C program. However, leaving sensitive data in memory increases the damage posed by memory disclosure vulnerabilities and direct attacks on physical memory. This leaves gap between what the standard considers correct and what a security developer might deem correct. Unfortunately, the C language does not provide a guaranteed way to achieve what the programmer intends, and attempts to add a memory scrubbing function to the C standard library have not seen mainstream adoption. Security-conscious developers have been left to devise their own means to keep the compiler from optimizing away their scrubbing functions, and this has led to a proliferation of "secure memset" implementations of varying quality.

The aim of this paper is to understand the current state of the dead store elimination and programmers' attempts to circumvent it. We begin with a survey of existing techniques used to scrub memory found in open source security projects. Among more than half a dozen techniques, we found that several are flawed and that none are both universally available and effective. Next, using a specially instrumented version of the Clang compiler, we analyzed eleven high-profile security projects to determine whether their implementation of a scrubbing function is effective and whether it is used consistently within the project. We found that only threeof the eleven projects did so.

To aid the current state of affairs, we developed a single best-of-breed scrubbing function that combines the effective techniques we found in our survey. We have shared our implementation with developers of the projects we surveyed that lacked a reliable scrubbing function and have made it available to the public. While not a perfect solution, we believe ours combines the best techniques available today and offers a developers a ready to use solution for their own projects.

We also developed a *scrubbing aware* C compiler based on Clang. Our compiler protects scrubbing operations by inhibiting dead store elimination in case where a store operation may have been intended as a scrubbing operation by the programmer. Our solution does not completely disable DSE, minimizing the performance impact of our mechanism. Our performance evaluation shows that our modified compiler introduces virtually no performance penalty.

In total, our contributions are as follows:

- We survey scrubbing techniques currently found in the wild, scoring each in terms of its *availability* and *reliability*. In particular, we identify several flawed techniques, which we reported to developers of projects relying on them. We also report on the performance of each technique, where we found an order of magnitude difference between the best and worst performing techniques.
- We present a case study of eleven security projects that have implemented their own scrubbing function. We found that no two projects' scrubbing functions use the same set of techniques. We also identify common pitfalls encountered in real projects.
- We develop and make publicly available a best-of-breed scrubbing function that combines the most reliable techniques found in use today.
- We develop a scrubbing-safe dead store elimination optimization that protects memory writes intended to scrub sensitive data from being eliminated. Our mechanism has negligible performance overhead and can be used without any source code changes.

The rest of the paper is organized as follows. Section 2 provides background for the rest of the paper and describes related work. Section 3 surveys the existing techniques that are used to implement reliable scrubbing functions and then Section 4 evaluates their performance. Section 5 examines the reliability and usage of scrubbing functions of eleven popular open source applications. Section 6 describes our `secure_memzero` implementation. Section 7 describes our secure DSE implementation and evaluates its performance. Section 8 discusses out results. Section 9 concludes the paper.

2 Background and Related Work

D'Silva *et al.* [29] use the term *correctness-security gap* to describe the gap between the traditional notion of compiler correctness and the correctness notion that a security-conscious developers might have. They found instances of a correctness-security gap in several optimizations, including dead store elimination, function inlining, code motion, common subexpression elimination, and strength reduction.

Lu *et al.* [31] investigate an instance of this gap in which the compiler introduces padding bytes in data structures to improve performance. These padding bytes may remain uninitialized and thus leak data if sent to the outside world. By looking for such data leakage, they found previously undiscovered bugs in the Linux and Android kernels. Wang *et al.* [38] explore another instance of the correctness-security gap: compilers sometimes remove code that has undefined behavior that, in some cases, includes security checks. They developed a static checker called STACK that identifies such code in C/C++ programs and they used it to uncover 160 new bugs in commonly deployed systems.

Our work examines how programmers handle the correctness-security gap introduced by aggressive dead store elimination. While the soundness and security of dead store elimination has been studies formally [27, 30, 28], the aim of our work is to study the phenomenon *empirically*.

Bug reports are littered with reports of DSE negatively affecting program security, as far back as 2002 from Bug 8537 in GCC titled "Optimizer Removes Code Necessary for Security" [3], to January 2016 when OpenSSH patched CVE-2016-0777 which allowed a malicious server to read private SSH keys by combining a memory disclosure vulnerability with errant `memset` and `bzero` memory scrubs [9]; or February 2016 when OpenSSL changed its memory scrubbing technique after discussion in Issue 445 [21]; or Bug 751 in OpenVPN from October 2016 about secret data scrubs being optimized away [25].

Despite developers' awareness of such problems, there is no uniformly-used solution. The CERT C Secure Coding Standard [36] recommends `SecureZeroMemory` as a Windows solution, `memset_s` as a C11 solution, and the volatile data pointer technique as a C99 solution. Unfortunately, each of these solutions has problems. The Windows solution is not cross-platform. For the recommended C11 `memset_s` solution, to the best of our knowledge, there is no standard-compliant implementation. Furthermore, while the CERT solution for C99 solution may prevent most compilers from removing scrubbing operations, the standard does not guarantee its correctness [35]. Furthermore, another common technique, using a volatile function pointer, is not guaranteed to work according to the standard because although the standard requires compilers to access the function pointer, it does not require them to make a call via that pointer [34].

3 Existing Approaches

Until recently, the C standard did not provide a way to ensure that a `memset` is not removed, leaving developers who wanted to clear sensitive memory were left to devise their own techniques. Here we survey eleven security-related open source projects to determine what techniques developers were using to clear memory. In this section, we present the results of our survey. For each technique, we describe how it is intended to work, its **availability** on different platforms, and its **effectiveness** at ensuring that sensitive data is scrubbed. 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.

In Section 4 we also compare the performance of a subset of the surveyed techniques.

The scrubbing techniques we found can be divided into four groups based on how to they attempt to force memory to be cleared:

- ◇ **Rely on the platform.** Use a function offered by the operating system or a library that guarantees memory will be cleared.
- ◇ **Disable optimization.** Disable the optimization that removes the scrubbing operation.
- ◇ **Hide semantics.** Hide the semantics of the clearing operation, preventing the compiler from recognizing it as a dead store.
- ◇ **Force write.** Directly force the compiler to write to memory.

In the remainder of this section, we describe and discuss each technique in detail and conclude with a performance evaluation of each technique. While performance is not the primary consideration when choosing a technique, it is, nevertheless, interesting to observe the wide performance disparity of each.

3.1 Platform-Supplied Functions

The easiest way to ensure that memory is scrubbed is to call a function that guarantees that memory will be scrubbed. These *deus ex machina* techniques rely on a platform-provided function that guarantees the desired behavior and lift the burden of fighting the optimizer from the developers' shoulders. Unfortunately, these techniques are not universally available, forcing developers to come up with backup solutions.

3.1.1 Windows SecureZeroMemory

On Windows, `SecureZeroMemory` is designed to be a reliable scrubbing function even in the presence of optimizations. This is achieved by the support from the Microsoft Visual Studio compiler, which never optimizes out a call to `SecureZeroMemory`. Unfortunately, this function is only available on Windows.
Used in: Kerberos's `zap`, Libsodium's `sodium_memzero`, Tor's `memwipe`.
Availability: Windows platforms.
Effectiveness: Effective.

3.1.2 OpenBSD explicit_bzero

Similarly OpenBSD provides `explicit_bzero`, a optimization-resistant analogue of the BSD `bzero` function. The `explicit_bzero` function has been available in OpenBSD since version 5.5 and FreeBSD since version 11. Under the hood, `explicit_bzero` simply calls `bzero`, however, because `explicit_bzero` is defined in the C standard library shipped with the operating system and not in the compilation unit of the program using it, the compiler is not aware of this and does not eliminate the call to `explicit_bzero`. As discussed in Section 3.3.1, this way of keeping the compiler in the dark only works if definition and use remain separate through compilation and linking. This is the case with OpenBSD and FreeBSD, which dynamically link to the C library at runtime.
Used in: Libsodium's `sodium_memzero`, Tor's `memwipe`, OpenSSH's `explicit_bzero`.
Availability: FreeBSD and OpenBSD.
Effectiveness: Effective (when `libc` is a shared library).

3.1.3 C11 memset_s

Annex K of the C standard (ISO/IEC 9899-2011) introduced the `memset_s` function, declared as

```
errno_t memset_s(void* s, rsize_t smax,
                 int c, rsize_t n);
```

Similar to `memset`, the `memset_s` function sets a number of the bytes starting at address `s`to the byte value `c`. The number of bytes written is the lesser of `smax` or `n`. By analogy to `strncpy`, the intention of having two size arguments is prevent a buffer overflow when `n` is an untrusted user-supplied argument; setting `smax` to the size allocated for `s` guarantees that the buffer will not be overflowed. More importantly, the standard requires that the function actually write to memory, regardless of whether or not the written values are read.

The use of two size arguments, while consistent stylistically with other `_s` functions, has drawbacks. It differs from the familiar `memset` function which takes one size argument. The use of two arguments means that a programmer can't use `memset_s` as a drop-in replacement for `memset`. It may also lead to incorrect usage, for example, by setting `smax` or `n` to 0, and thus, while preventing a buffer overflow, would fail to clear the buffer as intended.

While `memset_s` seems like the ideal solution, it's implementation has been slow. There may be several reasons for this. First, `memset_s` is not required by the standard. It is part of the optional Appendix K. C11 treats all the function in the Annex K as a unit. That

is, if a C library wants to implement the Annex K in a standard-conforming fashion, it has to implement *all* of the functions defined in this annex. At the time of this writing, `memset_s` is not provided by the GNU C Library nor by the FreeBSD, OpenBSD, or NetBSD standard libraries. It's poor adoption and perceived flaws have led to calls for its removal from the standard [32].
Used in: Libsodium's `sodium_memzero`, Tor's `memwipe`, OpenSSH's `explicit_bzero`, CERT's Windows-compliant solution [36].
Availability: No mainstream support.
Effectiveness: Effective.

3.2 Disabling Optimization

Since the scrubbing store elimination problem is caused by compiler optimization, it is possible to prevent scrubbing stores from being eliminated by disabling compiler optimization. Dead store elimination is enabled (on GCC and Clang) at optimization level `-O1`, so code compiled with no optimization would retain the scrubbing writes. However, disabling optimization completely can significantly degrade performance, and is eschewed by developers. Alternatively, some compilers allow optimizations to be enabled individually, so, in principle, a program could be compiled with all optimizations except dead store elimination enabled. However, some optimization passes work better when dead stores have already been eliminated. Also, specifying the whole list of optimization passes instead of a simple optimization level like `O2` is cumbersome.

Many compilers, including Microsoft Visual C, GCC and Clang, provide built-in versions of some C library functions, including `memset`. During compilation, the compiler replaces calls to the C library function with its built-in equivalent to improve performance. In at least one case we found, developers attempted to preserve scrubbing stores by disabling the built-in `memset` intrinsic using the `-fno-builtin-memset` flag. Unfortunately, while this may disable the promotion of standard C library functions to intrinsics, it does not prevent the compiler from understanding the semantics of `memset`. Furthermore, as we found during our performance measurements (Section 4), the `-fno-builtin-memset` flag does not not prevent the developer from calling the intrinsic directly, triggering dead store elimination. In particular, starting with glibc 2.3.4 on Linux, defining `_FORTIFY_SOURCE` to be an integer greater than 0 enables additional compile-time bounds checks in common functions like `memset`. In this case, if the checks succeed, the inline definition of `memset` simply calls the built-in `memset`. As a result, the `-fno-builtin-memset` option did not protect scrubbing stores from dead store elimination.
Used in: We are not aware of any programs using this technique.
Availability: Widely available.
Effectiveness: Flawed (not working when newer versions of glibc and GCC are used and optimization level is `O2` or `O3`).

3.3 Hiding Semantics

Several scrubbing techniques attempt to hide the semantics of the scrubbing operation from the compiler. The thinking goes, if the compiler doesn't recognize that an operation is clearing memory, it will not remove it.

3.3.1 Separate Compilation

The simplest way to hide the semantics of a scrubbing operation from the compiler is to implement the scrubbing operation (e.g. by simply calling `memset`) in a separate compilation unit. When this scrubbing function is called in a different compilation unit than the defining one, the compiler cannot remove any calls to the scrubbinf function because the compiler does not know that it is equivalent to `memset`. Unfortunately, this technique is not reliable when link-time optimization (LTO) is enabled, which can merge all the compilation units into one, giving the compiler a global view of the whole program. The compiler can then recognize that the scrubbing function is effectively a `memset`, and remove it. Thus, to ensure this technique works, the developer needs to make sure that she has the control over how the program is compiled.

3.3.2 Weak Linkage

GCC and some compilers that mimic GCC allow developers to define *weak definitions*. A weak definition of a symbol, indicated by the compiler attribute `__attribute__((weak))`, is a tentative definition that may be replaced by another definition at link time. In fact, the OpenBSD `explicit_bzero` function (Section 3.1.2) uses this technique also:

```
__attribute__((weak)) void
__explicit_bzero_hook(void *buf, size_t len) { }

void explicit_bzero(void *buf, size_t len) {
    memset(buf, 0, len);
    __explicit_bzero_hook(buf, len);
}
```

The compiler can not eliminate the call to `memset` because an overriding definition of `__explicit_bzero_hook` may access `buf`. This way, even if `explicit_bzero` is used in the same compilation unit where it is defined, the compiler will not eliminate the scrubbing operation. Unfortunately, this technique is also vulnerable to link-time optimization. With link-time optimization enabled, the compiler-linker can resolve the final definition of the weak symbol, determine that it does nothing, and then eliminate the

dead store.
Used in: Libsodium's sodium_memzero, libressl's explicit_bzero [13].
Availability: Available on GCC and Clang.
Effectiveness: Flawed (defeated by LTO).

3.3.3 Volatile Function Pointer

Another popular technique for hiding a scrubbing operation from the compiler is to call the memory scrubbing function via a volatile function pointer. OPENSSL_cleanse of OpenSSL 1.0.2, shown below, is one implementation that uses this technique:

```
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);
}
```

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.

The effect of declaring memset_func as volatile means that the compiler must read its value from memory each time its used because the value may have changed. The reasoning goes that because the compiler does not know the value of memset_func at compile time, it can't recognize the call to memset and eliminate it.

We have confirmed that this technique works on GCC, Clang and Microsoft Visual C, and we deem it to be effective. It is worth noting, however, that while the standard requires the compiler to read the value of memset_func from memory, it does *not* require it to call memset if it can compute the same result by other means. Therefore, a compiler would be in compliance if it inlined each call to OPENSSL_cleanse as:

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

If the memory pointed to by ptr is not read again, then the direct call to memset, the semantics of which are known, could be eliminated, removing the scrubbing operation. We know of no compiler that does this and consider such an optimization unlikely.
Used in: OpenSSL 1.0.2's OPENSSL_cleanse (also used in Tor and Bitcoin); OpenSSH's explicit_bzero, quarkslab's memset_s [4].
Availability: Universally available.
Effectiveness: Effective in practice.

3.3.4 Assembly Implementation

Because optimizations often take place at compiler's intermediate representation level, it is possible to hide the semantics of a memory scrubbing operation by implementing it in assembly language. In some cases, this may also be done as a way to improve performance, however, our results indicate that the compiler's built-in intrinsic memset performs as well as the assembly implementation we examined. So long as the compiler does not perform assembly-level link-time optimization, this technique is effective at ensuring scrubbing stores are preserved.
Used in: OpenSSL's OPENSSL_cleanse (also used by Tor and Bitcoin); Crypto++'s SecureWipeBuffer.
Availability: Target-specific.
Effectiveness: Effective.

3.4 Forcing Memory Writes

The fourth set of techniques we found attempts to force the compiler to include the store operation without hiding its nature.

3.4.1 Complicated Computation

Several related techniques attempt to force the compiler to overwrite sensitive data in memory by forcing the compiler to carry out a computation. OPENSSL_cleanse from OpenSSL prior to version 1.0.2 is one example:

```
unsigned char cleanse_ctr = 0;
void OPENSSL_cleanse(void *ptr, size_t len) {
    unsigned char *p = ptr;
    size_t loop = len, ctr = cleanse_ctr;

    if (ptr == NULL) return;

    while (loop--) {
        *(p++) = (unsigned char)ctr;
        ctr += (17 + ((size_t)p & 0xF));
    }
    p = memchr(ptr, (unsigned char)ctr, len);
    if (p) ctr += (63 + (size_t)p);
    cleanse_ctr = (unsigned char)ctr;
}
```

This function reads and writes the global variable cleanse_ctr, which provides varying garbage data to fill the memory to be cleared. Because accesses to the global variable have a global impact on the program, the compiler cannot determine that this function is useless without extensive interprocedural analysis. Since such interprocedural analysis is expensive, the compiler most likely does not perform it, thus it cannot figure out that `OPENSSL_cleanse` is actually a scrubbing function. However, this particular implementation is notoriously slow (see the performance numbers in Section 4). OpenSSL gave up this technique in favor of the volatile function pointer technique (Section 3.3.3) starting with version 1.0.2.

Another way to scrub sensitive data is to simply rerun the computation that accesses sensitive data again. This is used in the musl libc [16] implementation of `bcrypt`, which is a popular password hashing algorithm. musl's `bcrypt` implementation `__crypt_blowfish` calls the hashing function `BF_crypt` twice: the first time it passes the actual password to get the hash, the second time it passes a test password. The second run serves two purposes. First, it is a self-test of the hashing code. `__crypt_blowfish` compares the result of the second run with the hardcoded hash value in the function. If they do not match, there is something wrong in the hashing code. (In fact, the developers of musl libc found a bug in GCC that manifested in their hashing code [10].) Second, the second run of `BF_crypt` can also clear sensitive data left on the stack or in registers by the first run. Since the same function is called twice, the same registers will be used, thus the sensitive data left in registers will be cleared. Since the two calls to `BF_crypt` are in the same scope and the stack pointer points to the same position of the stack before the two calls, the sensitive data left on the stack by the first run should be cleared by the second run. The advantage of this solution is that it clears sensitive data not only on the stack but also in registers.

While the complicated computation technique appears effective in practice, there is no guarantee that a compiler will not someday see through the deception. This technique, especially re-running the computation, has a particularly negative performance impact.
Used in: `OPENSSL_cleanse` from OpenSSL 1.0.1 (also used in Tor and Bitcoin), `crypt_blowfish` from musl libc [16].
Availability: Universal.
Effectiveness: Effective in practice.

3.4.2 Volatile Data Pointer

Another way to force the compiler to perform a store is to access a volatile-qualified type. As noted in Section 3.3.3, the standard requires accesses to objects that have volatile-qualified types to be performed explicitly. If the memory to be scrubbed is a volatile object, the compiler will be forced to preserve stores that would otherwise be considered dead. Cryptography Coding Standard's `Burn` is one of the implementations based on this idea:

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

In the function above, the memory to be scrubbed is written via a pointer-to-volatile `p` in the while loop. We have found that this technique is effective on GCC, Clang, and Microsoft Visual C. Unfortunately, this behavior is *not* guaranteed by the C11 standard: "What constitutes an access to an object that has volatile-qualified type is implementation-defined." This means that, while accessing an object declared volatile is clearly an "access to an object that has volatile-qualified type" (as in the case of the function pointer that is a volatile object), accessing a non-volatile object via pointer-to-volatile may or may not be considered such an access.
Used in: `sodium_memzero` from Libsodium, `insecure_memzero` from Tarsnap, `wipememory` from Libgcrypt, `SecureWipeBuffer` from the Crypto++ library, `burn` from Cryptography Coding Standard [37], David Wheeler's `guaranteed_memset` [39], `ForceZero` from wolfSSL [26], `sudo_memset_s` from sudo [22], and CERT's C99-compliant solution [36].
Availability: Universal.
Effectiveness: Effective in practice.

3.4.3 Memory Barrier

Both GCC and Clang support a memory barrier expressed using an inline assembly statement. 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 [1]. This indicates to the compiler that the inline assembly statement may access and modify memory, forcing it to keep stores that might otherwise be considered dead. GCC's documentation indicates that the following inline assembly should work as a memory barrier [1]:

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

Our testing shows the above barrier works with GCC, and since Clang also supports the same syntax, one would expect that the barrier above would also work with Clang. In fact, it may remove a `memset` call before such a barrier [6]. We found that Kerberos (more in Section 5.2) uses this barrier to implement its scrubbing function, which may be unreliable with Clang. A more reliable way to define memory barrier is illustrated by Linux's `memzero_explicit` below:

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

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

The difference is the `"r"(ptr)` argument, which makes the pointer to the scrubbed memory visible to the assembly code and prevents the scrubbing store from being eliminated.
Used in: `zap` from Kerberos, `memzero_explicit` from Linux [15].
Availability: Clang and GCC.
Effectiveness: Effective in practice.

3.5 Discussion

Our survey of existing techniques indicates that **there is no single best technique for scrubbing sensitive data**. The most effective techniques are those where the integrity of scrubbing operation is guaranteed by the platform. Unfortunately, this means that creating a scrubbing function requires relying on platform-specific functions rather than a standard C library or POSIX function.

Of the remaining techniques, we found that the volatile data pointer, volatile function pointer, and compiler memory barrier techniques are *effective in practice* with the compilers we tested. The first two of these, relying on the volatile storage type, can be used with any compiler but are not guaranteed by the standard. The memory barrier technique is specific to GCC and Clang and its effectiveness may change without notice as it has done already.

4 Performance

When it comes to security-sensitive operations like data scrubbing, performance is a secondary concern. Nevertheless, given two equally good choices, one would prefer one that is more efficient. In this section, we present our results of benchmarking the scrubbing techniques we described above under Clang 3.9 and GCC 6.2. Our baseline is the performance of ordinary `memset`, both the C library implementation and the built-in intrinsics in Clang and GCC. The performance of the C library implementation represents the expected performance of non-inlined platform-provided solutions (Section 3.1) and the separate compilation (Section 3.3.1) and weak linkage (Section 3.3.2) techniques without link-time optimization. The performance of GCC and Clang intrinsics represents the expected performance of inlined platform-provided solutions (Section 3.1) as well as the memory barrier technique (Section 3.4.3), assuming the scrubbing function is inlined. We also measured the performance of the volatile function pointer technique (Section 3.3.3), the volatile data pointer technique (Section 3.4.2), the custom assembly implementation of OpenSSL 1.1.0b (Section 3.3.4), and the complicated computation technique of OpenSSL prior to version 1.0.2 (Section 3.4.1).

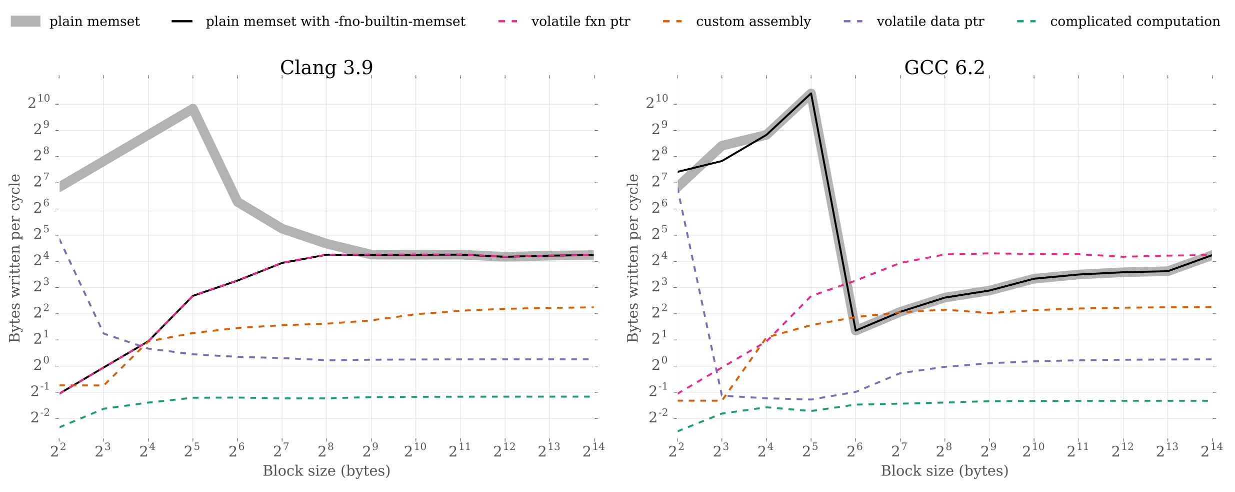
4.1 Methodology

We compiled a unique executable for each technique and block size on GCC 6.2 and Clang 3.9 with the `-O2` option targeting the x86_64 platform. A scrubbing routine's performance is the median runtime over 16 program executions, where each execution gives the median runtime over 256 trials, and each trial gives the mean runtime of 256 scrubbing calls. Program executions for a given test case were spaced out in order to eliminate any affects caused by the OS scheduler interrupting a particular program execution. We left the testing framework code unoptimized. Scrubbing calls were followed by inline assembly barriers to ensure that optimizations to scrubbing routines did not affect benchmarking code. The benchmarking code calls a generic scrub function, which then calls the specific scrubbing routine to be tested; this code is allowed to be optimized, so as a result the scrubbing routine is typically inlined within the generic scrub function. The scrubbing function and scrubbed buffer size are defined at compile time, so optimizations can be exhaustive. The time to iterate through a loop 256 times containing a call to a no-op function and memory barrier was subtracted from each trial in order to eliminate time spent executing benchmarking code and the generic scrub function call. The runtime for a scrubbing routine was calculated with the `rdtsc` and `rdtscp` instructions which read the time stamp counter, with the help of the `cpuid` instruction which serializes the CPU and thus ensures that no other code is benchmarked [33]. Instruction and data caches were warmed up by executing the benchmarking code 4 times before results were recorded. Program executions were tied to the same CPU core to ensure that consistent hardware was used across tests.

The tests were done on an Intel Xeon E5-2430 v2 processor with x86_64 architecture and a 32KB L1d cache, 32KB L1i cache, and 256K L2 cache running Ubuntu 14.04 with Linux kernel 3.13.0-100-generic.

4.2 Results

Figures 1 shows the results of our benchmarks. The left plot (Figure 1a) shows the result of compiling each technique using Clang 3.9, the right plot (Figure 1b) shows the result of compiling each technique using GCC 6.2. In each plot, the *x*-axis shows the block size being zeroed and the *y*-axis the bytes written per cycle, computed by dividing the number of cycles taken by the block size. The heavy solid grey line shows the performance of plain `memset` when it is not removed by the optimizer. The fine solid black line is performance of plain `memset`



(a) Compiled with Clang 3.9.

(b) Compiled with GCC 6.2.

Figure 1: Performance of various scrubbing implementations compiled at optimization level `-O2`. The *x*-axis shows the block size being zeroed and the *y*-axis the bytes written per cycle, computed by dividing the number of cycles taken by the block size.

when compiled with the `-fno-builtin-memset` option, which instructs the compiler not to use its own built-in intrinsic `memset` instead of calling the C standard library implementation. The remaining dashed lines show the performance of the volatile function pointer technique (red line), the custom assembly implementation from OpenSSL (orange line), the volatile data pointer technique (blue line), and the complicated computation technique from OpenSSL (green line).

Large block sizes. At large block sizes, performance is dominated by the efficiently of each implementation. The largest determining factor of an implementation's efficiency is the size of its move instructions: "plain memset" and "volatile function pointer" both jump to libc's `memset`, which performs a loop of `movdqa` instructions (2^4 bytes/instruction); "custom assembly" performs a loop of `movq` instructions (2^3 bytes/instruction); and "volatile data pointer" performs a loop of `movb` instructions (2^0 byte/instruction). Further, "complicated computation" performs several unnecessary obfuscating instructions in order to trick the compiler. Its poor performance reflects the numerous developers reports complaining about its slow speed, for example Tor Ticket #7419 titled "Choose a faster memwipe implementation" [2].

Additionally, implementations which align the block pointer see improved efficiency. Libc's `memset` is able to perform `movdqa` instructions after it dqword-aligns its pointer. "custom assembly" improves from 2^3 to 2^4 byte block sizes because above that threshold it qword-aligns its pointer in order to perform `movq` instructions.

Furthermore, at some point ($\geq 2^9$ bytes for Clang; $\geq 2^{14}$ bytes for GCC) the built-in `memset` defers to using libc's `memset`, hence it is identical to "volatile function pointer" given large block sizes.

Small block sizes. At small block sizes, performance is dominated by whether or not loop unrolling occurred. The scrubbing routine is given the block size at compile-time, so it is able to optimize accordingly. Thus, for "plain memset", move instructions are unrolled for sizes $\leq 2^8$ bytes on Clang and sizes $\leq 2^5$ bytes on GCC. Additionally, for the "volatile data pointer" technique, unrolling occurs for sizes $\leq 2^6$ bytes on Clang and sizes $\leq 2^2$ bytes on GCC. Note that the performance of implementations' unrolled loops are different because different types of move instructions may be unrolled (such as a `movb` versus a `movq`).

The large magnitude of spikes in the graph can be attributed to the superscalar nature of the CPU it is run on, which essentially gives it those instructions for free for small block sizes. Both Clang and GCC-compiled "plain memset" code see a major performance drop between 32- and 64-byte block sizes. Although for GCC, this is the point at which unrolling no longer occurs—it is not so for Clang, whose dropoff is less severe. We suspect this is due to L1 caching of smaller size blocks. (The L1 cache line size is 64 bytes on our architecture.)

GCC's builtin. Upon first examining our results, we were surprised to find that the GCC-compiled "plain memset" with `-fno-builtin-memset` did as well as "plain memset" with the built-in intrinsic `memset`. After examining the produced assembly code, we found that the scrubbing function was *not* calling the libc `memset` function as expected (and the Clang-compiled version was). As a result, we found that `string.h` (where `memset` is declared) changes its behavior based on the value of the _FORTIFY_SOURCE macro, as described in Section 3.2. Thus, even with the -fno-builtin-memset option, GCC generated its built-in memset. Under normal circumstances, such code would be subject to dead-store elimination, causing the scrubbing operation to be removed.

4.3 Discussion

Our performance measurements found that **techniques vary drastically in performance**. This may make some techniques preferable to others.

5 Case Studies

To understand the use of memory scrubbing in practice, we examined the eleven popular security libraries and applications listed in Table 1. Our choices were guided by whether or not the code handled sensitive data (e.g. secret keys), availability of the source code and our own judgement of the project's relevance. For each project, we set out to determine whether a memory scrubbing function is **available**, **effective**, and **used consistently** by the projects' developers. We used the latest stable version of each project as of October 9, 2016.

Availability. To determine whether a scrubbing function is available, we manually examined the program source code. All eleven projects used one or more of the techniques described in Section 3 to clear sensitive data, and seven of them relied on a combination of at least two techniques.

If a project relied on more than one technique, it automatically chose and used the first technique available on the platform in order of preference specified by the developer. Columns under the *Preference* heading in Table 1 show the developer preference order for each technique, with 1 being highest priority (first chosen if available). The scrubbing techniques listed under the *Preference* heading are: *Win* is Windows' SecureZeroMemory, *BSD* is BSD's explicit_bzero, *C11* is C11's memset_s, *Asm.* is a custom assembly implementation, *Barrier* is the memory barrier technique, *VDP* is the volatile data pointer technique, *VFP* is the volatile function pointer technique, *Comp.* is the complicated computation technique, *WL* is the weak linkage technique, and *memset* is a call to plain memset. If a project used a function that can be one of many techniques depending on the version of that function—for example, projects that use OpenSSL's OPENSSL_cleanse, which may either be *VFP* or *Comp.* depending on if OpenSSL version $\geq$1.0.2 or $<$1.0.2 is used—the newer version is given a higher preference. An $*$ indicates an incorrectly implemented technique.

For example, Tor uses Windows' SecureZeroMemory if available, then BSDs' explicit_bzero if available, and so on. Generally, for projects that used them, all chose a platform-supplied function (SecureZeroMemory, explicit_bzero, or memset_s) first before falling back to other techniques. The most popular of the do-it-yourself approaches are the volatile data pointer (VDP) and volatile function pointer (VFP) techniques, with the latter being more popular with projects that attempt to use a platform-provided function first.

Effectiveness. To answer the second question—whether the scrubbing function is effective—we relied on the manual analysis in Section 3. If a project used an unreliable or ineffective scrubbing technique in at least one possible configuration, we considered its scrubbing function ineffective, and scored it *flawed*, denoted ○ in the *Score* column. If the scrubbing function was effective and used consistently, we scored it *effective*, denoted ●. If it was effective but not used consistently, we scored it *inconsistent*, denoted ◐.

Consistency. To determine whether a function was used consistently, 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. We did not report writes that were eliminated because they were followed by another write to the same memory location, because in this case, the data would be cleared by the second write. Additionally, if sensitive data is small enough to be fit into registers, it may be promoted to a register, which will lead to the removal of the scrubbing store [1]. Since the scrubbing store is not removed in the dead store elimination pass, our tool does not report it. We would argue such removals have less impact on security since the sensitive data is in a register. However, if that register spilled when the sensitive data in it, it may still leave some sensitive data in memory. Appendix A.1 provides additional details of our tool. We compiled each project using this compiler with the same optimization options as in the default build of the project. Then we examined the report generated by our tool and manually identified cases of dead store elimination that removed scrubbing operations.

Of the eleven projects we examined, all of them supported Clang. We note, however, that our goal in this part of our analysis is to identify sites where a compiler *could* eliminate a scrubbing operation, and thus identify sites where sensitive variables were not being cleared as intended by the developer. We then examined each case to determine whether the memory contained sensitive data, and whether dead store elimination took place because a project's own scrubbing function was not used or because the function was ineffective. If cases of the latter,

[1] For example, at the end of OpenSSH's SHA1Transform function, "a=b=c=d=e=0;" is used to scrub sensitive data. Because all the five variables are in virtual registers in the IR form, no store is eliminated in the DSE pass.

	Removed ops.					*Preference*										*Score*
Project	*Total*	*Sensitive*	*Heap*	*Stack*	*H/S*	*Win*	*BSD*	*C11*	*Asm.*	*Barrier*	*VDP*	*VFP*	*Comp.*	*WL*	*memset*	
NSS	15	9	3	12	0	-	-	-	-	-	-	-	-	-	1	○
OpenVPN	8	8	2	6	0	-	-	-	-	-	-	-	-	-	1	○
Kerberos	10	2	9	0	1	1	-	-	-	2*	-	-	-	-	3	○
Libsodium	0	0	0	0	0	1	3	2	-	-	5	-	-	4	-	○
Tarsnap	11	10	10	1	0	-	-	-	-	-	1	-	-	-	-	◐
Libgcrypt	2	2	0	2	0	-	-	-	-	-	1	-	-	-	-	◐
Crypto++	1	1	0	1	0	-	-	-	1	-	2	-	-	-	-	◐
Tor	4	0	4	0	0	1	2	3	4	-	-	5	6	-	-	◐
Bitcoin	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●
OpenSSH	0	0	0	0	0	-	1	2	-	-	-	3	-	-	-	●
OpenSSL	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●

Table 1: Summary of open source projects' removed scrubbing operations and the scrubbing techniques they use. *Removed ops.* columns show the total number of removed scrubs, the number of removed scrubs dealing with sensitive data, and the locations of memory that failed to be scrubbed. *Preference* columns show the developer preference order for each technique, with 1 being highest priority (first chosen if available). The $*$ in the row for Kerberos indicates that its barrier technique was not implemented correctly; see Section 3.4.3 for discussion. A project's *Score* shows whether its scrubbing implementation is *flawed* (○), *inconsistent* (◐), or *effective* (●).

we determined why the function was not effective; these findings are reflected in the results reported in Section 3. Columns under the heading *Removed ops.* in Table 1 show the number of cases where a scrubbing operation was removed. The *Total* column shows the total number of sites where an operation was removed. The *Sensitive* column shows the number of such operations where we considered the data to be indeed sensitive. (In some cases, the scrubbing function was used to clear data that we did not consider sensitive, such as pointer addresses.) The *Heap*, *Stack*, and *H/S* columns indicate whether or not the cleared memory was allocated on the heap, on the stack stack, or potentially on either heap or stack.

Of the eleven projects examined, four had an effective scrubbing function but did not use it consistently, resulting in a score of *inconsistent*, denoted ◐ in Table 1. As the results in Table 1 show, **only three of the eleven projects had a scrubbing function that was effective and used consistently.**

We notified the developers of each project that we scored *flawed* or *inconsistent*. For our report to the developers, we manually verified each instance where a scrubbing operation was removed, reporting only valid cases to the developers. Generally, as described below, developers acknowledged our report and fixed the problem. Note that none of the issues resulted in CVEs because to exploit, they must be used in conjunction with a separate memory disclosure bug and these types of bugs are outside the scope of this work.

In the remainder of this section, we report on the open source projects that we analyzed. Our goal is to identify common trends and understand how developers deal with the problem of compilers removing scrubbing operations.

5.1 OpenVPN

OpenVPN is an TLS/SSL-based user-space VPN [20]. We tested version 2.3.12. OpenVPN 2.3.12 does not have a reliable memory scrubbing implementation since it uses a `CLEAR` macro which expands to `memset`. We found 8 scrubbing operations that were removed, all of which deal with sensitive data. Each of the removed operations used `CLEAR`, which is not effective.

Sample case. Function `key_method_1_read` in Figure 2 is used in OpenVPN's key exchange function to process key material received from an OpenVPN peer. However, the `CLEAR` macro fails to scrub the key on the stack since it is a call to plain `memset`.

Developer response. The issues were reported, although OpenVPN developers were already aware of the problem and had a ticket on their issue tracker for it that was opened 12 days prior to our notification [25]. The patch does not change the `CLEAR` macro since it is used extensively throughout the project, but it does replace many `CLEAR` calls with our recommended fix discussed in Section 6 [7].

5.2 Kerberos

Kerberos is a network authentication protocol that provides authentication for client/server applications by using secret-key cryptography [11]. We tested Kerberos release krb5-1.14.4. The Kerberos memory scrubbing im-

```
/* From openvpn-2.3.12/src/openvpn/basic.h */
#define CLEAR(x) memset(&(x), 0, sizeof(x))

/* From openvpn-2.3.12/src/openvpn/ssl.c */
static bool key_method_1_read (struct buffer *buf, struct
  tls_session *session) {

  struct key key;
  /* key is allocated on stack to hold TLS session key */
  ...
  /* Clean up */
  CLEAR (key);
  ks->authenticated = true;
  return true;
}
```

Figure 2: A removed scrubbing operation in OpenVPN 2.3.12.

plementation, zap, is unreliable. First, it defaults to Windows' SecureZeroMemory, which is effective. Otherwise it uses a memory barrier that may not prevent the scrubbing operation from being removed when the code is compiled with Clang (see Section 3.4.3). Finally, if the compiler is not GCC, it uses a function that calls memset. While this is more reliable than a macro, memset may be removed if LTO is enabled (see Section 3.3.1). Furthermore, even though zap is available (and reliable on Windows), plain memset is still used throughout the code to perform scrubbing. We found 10 sites where scrubbing was done using memset, which is not effective; 2 of these sites deal with sensitive data.

Sample case. Function free_lucid_key_data in Figure 3 is used in Kerberos to free any storage associated with a lucid key structure (which is typically on the heap) and to scrub all of its sensitive information. However it does so with a call to plain memset, which is then removed by the optimizer.

Developer response. The issues have been patched with calls to zap. In addition, zap has been patched according to our recommended fix discussed in Section 6.

```
static void free_lucid_key_data(gss_krb5_lucid_key_t *key) {
  if (key) {
    if (key->data && key->length) {
      memset(key->data,0,key->length);
      xfree(key->data);
      memset(key,0,sizeof(gss_krb5_lucid_key_t));
    }
  }
}
```

Figure 3: A removed scrubbing operation in Kerberos release krb5-1.14.4.

5.3 Tor

Tor provides anonymous communication via onion routing [24]. We tested version 0.2.8.8. Tor defines memwipe, which reliably scrubs memory: it uses Windows' SecureZeroMemory if available, then RtlSecureZeroMemory if available, then BSD's explicit_bzero, then memset_s, and then OPENSSL_cleanse, which is described below. Despite the availability of memwipe, Tor still uses memset to scrub memory in several places. We found 4 scrubbing operations that were removed, however none dealt with sensitive data.

Sample case. Function MOCK_IMPL in Figure 4 is used to destroy all resources allocated by a process handle. However, it scrubs the process handle object with memset, which is then removed by the optimizer.

Developer response. The bugs were reported and have yet to be patched.

```
MOCK_IMPL(void, tor_process_handle_destroy,(process_handle_t
  *process_handle, int also_terminate_process)) {

  /* process_handle is passed in and allocated on heap to
   * hold process handle resources */
  ...
  memset(process_handle, 0x0f, sizeof(process_handle_t));
  tor_free(process_handle);
}
```

Figure 4: A removed scrubbing operation in Tor 0.2.2.8.

5.4 OpenSSL

OpenSSL is a popular TLS/SSL implementation as well as a general-purpose cryptographic library [19]. We tested version 1.1.0b. OpenSSL uses OPENSSL_cleanse to reliably scrub memory. OPENSSL_cleanse defaults to its own assembly implementations in various architectures unless specified otherwise by the no-asm flag at configuration. Otherwise, starting with version 1.0.2, it uses the volatile function pointer technique to call memset. Prior to version 1.0.2, it used the complicated computation technique. We found no removed scrubbing operations in version 1.1.0b.

5.5 NSS

Network Security Services (NSS) is an TLS/SSL implementation that traces its origins to the original Netscape implementation of SSL [17]. We tested version 3.27.1. NSS does not have a reliable memory scrubbing implementation since it either calls memset or uses the macro PORT_Memset, which expands to memset. We found 15 scrubbing operations that were removed, 9 of which deal with sensitive data. Of the 15 removed operations, 6 were calls to PORT_Memset and 9 were calls to plain memset.

Sample case. Function PORT_ZFree is used throughout the NSS code for freeing sensitive data and is based on function PORT_ZFree_stub in Figure 5. However PORT_ZFree_stub's call to memset fails to scrub the pointer it is freeing.

Developer response. The bugs have been reported and Mozilla Security forwarded them to the appropriate team, however they have not yet been patched.

5.6 Libsodium

Libsodium is a cross-platform cryptographic library [14]. We tested version 1.0.11. Libsodium defines

```
extern void PORT_ZFree_stub(void *ptr, size_t len) {
  STUB_SAFE_CALL2(PORT_ZFree_Util, ptr, len);
  memset(ptr, 0, len);
  return free(ptr);
}
```

Figure 5: A removed scrubbing operation in NSS 3.27.1.

sodium_memzero, which does not reliably scrub memory. First, it defaults to Windows' SecureZeroMemory, then memset_s, and then BSD's explicit_bzero if available, which are all reliable. Then if weak symbols are supported, it uses a technique based on weak linkage, otherwise it uses the volatile data pointer technique. Techniques based on weak linkage are not reliable, because they can be removed during link-time optimization. All memory scrubbing operations used sodium_memzero, and since Libsodium is not compiled with link-time optimization, no scrubbing operations using sodium_memzero were removed.

5.7 Tarsnap

Tarsnap is a online encrypted backup service whose client source code is available [23]. We tested version 1.0.37. Tarsnap's memory scrubbing implementation, called insecure_memzero, uses the volatile data pointer scrubbing technique. Although insecure_memzero is an effective scrubbing function, Tarsnap does not use it consistently. We found 10 cases where memset was used to scrub memory instead of insecure_memzero in its keyfile.c, which handles sensitive data.

Sample case. Function read_encrypted in Figure 6 attempts to scrub a buffer on the heap containing a decrypted key. It is used throughout the project for reading keys from a Tarsnap key file. However, instead of using insecure_memzero, it uses plain memset, and is thus removed by the optimizer.

Developer response. Out of the eleven reported issues, the 10 in keyfile.c were already patched on July 2, 2016 but were not in the latest stable version. The one non-security issue does not require a patch, since the removed memset was redundant as insecure_memzero is called right before it.

```
static int read_encrypted(const uint8_t * keybuf, size_t
  keylen, uint64_t * machinenum, const char * filename,
  int keys) {

  uint8_t * deckeybuf;
  /* deckeybuf is allocated on heap to hold decrypted key */
  ...
  /* Clean up */
  memset(deckeybuf, 0, deckeylen);
  free(deckeybuf);
  free(passwd);
  free(pwprompt);
  return (0);
}
```

Figure 6: A removed scrubbing operation in Tarsnap 1.0.37.

5.8 Libgcrypt

Libgcrypt is a general purpose cryptographic library used by GNU Privacy Guard, a GPL-licensed implementation of the PGP standards [12]. We tested version 1.7.3. Libgcrypt defines wipememory, which is a reliable way of scrubbing because it uses the volatile data pointer technique. However, despite wipememory's availability and reliability, memset is still used to scrub memory in several places. We found 2 cases where scrubs were removed, and for both, memset is used to scrub sensitive sensitive data instead of wipememory.

Sample case. Function invert_key in Figure 7 is used in Libgcrypt's IDEA implementation to invert a key for its key setting and block decryption routines. However, invert_key uses memset to scrub a copy of the IDEA key on the stack, which is removed by the optimizer.

Developer response. The bugs have been patched with calls to wipememory.

```
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));
}
```

Figure 7: A removed scrubbing operation in Libgcrypt 1.7.3.

5.9 Crypto++

Crypto++ is a C++ class library implementing several cryptographic algorithms [8]. We tested version 5.6.4. Crypto++ defines SecureWipeBuffer, which reliably scrubs memory by using custom assembly if the buffer contains values of type byte, word16, word32, or word64; otherwise it uses the volatile data pointer technique. Despite the availability of SecureWipeBuffer, we found one scrubbing operation dealing with sensitive data that was removed because it used plain memset rather than its own SecureWipeBuffer.

Sample case. The UncheckedSetKey function, shown in Figure 8, sets the key for a CAST256 object. UncheckedSetKey uses plain memset to scrub the user key on the stack, which is removed by the optimizer.

Developer response. The bug was patched with a call to SecureWipeBuffer.

```
void CAST256::Base::UncheckedSetKey(const byte *userKey,
  unsigned int keylength, const NameValuePairs &) {

  AssertValidKeyLength(keylength);
  word32 kappa[8];
  /* kappa is allocated on stack to hold user key */
  ...
  memset(kappa, 0, sizeof(kappa));
}
```

Figure 8: A removed scrubbing operation in Crypto++ 5.6.4.

5.10 Bitcoin

Bitcoin is a cryptocurrency and payment system [5]. We tested version 0.13.0 of the Bitcoin client. The project defines `memory_cleanse`, which reliably scrubs memory by using `OPENSSL_cleanse`, described below. The source code uses `memory_cleanse` consistently; we found no removed scrubbing operations.

5.11 OpenSSH

OpenSSH is a popular implementation of the SSH protocol [18]. We tested version 7.3. OpenSSH defines its own `explicit_bzero`, which is a reliable way of scrubbing memory: it uses BSD's `explicit_bzero` if available, then `memset_s` if available. If neither are available, it uses the volatile function pointer technique to call `bzero`. We found no removed scrubbing operations.

5.12 Discussion

Our case studies lead us to two observations. First, **there is no single accepted scrubbing function.** Each project mixes its own cocktail using existing scrubbing techniques, and there is no consensus on which ones to use. Unfortunately, as we discussed in Section 3, some of the scrubbing techniques are flawed or unreliable, making scrubbing functions that rely on such techniques potentially ineffective. To remedy this state of affairs, we developed a single memory scrubbing technique that combines the best techniques into a single function, described in Section 6.

Second, even when the project has reliable scrubbing function, **developers do not use their scrubbing function consistently.** In four of the eleven projects we examined, we found cases where developers called `memset` instead of their own scrubbing function. To address this, we developed a scrubbing-safe dead-store elimination pass that defensively compile bodies of code, as discussed in Section 7.

6 Universal Scrubbing Function

As we saw in Section 3, there is no single memory scrubbing technique that is both universal and guaranteed. In the next section, we propose a compiler-based solution based on Clang, that protects scrubbing operations from dead-store elimination. In many cases, however, the developer can't mandate a specific compiler and must resort to imperfect techniques to protect scrubbing operations from the optimizer. To aid developers in this position, we developed our own scrubbing function, called `secure_memzero`, that combines the best effective scrubbing techniques in a simple implementation. Specifically, our implementation supports:

- ❖ Platform-provided scrubbing functions (`SecureZeroMemory` and `memset_s`) if available,
- ❖ The memory barrier technique if GCC or Clang are used to compile the source, and
- ❖ The volatile data pointer technique and the volatile function pointer technique.

Our `secure_memzero` function is implemented in a single header file `secure_memzero.h` that can be included in a C/C++ source file. The developer can specify an order of preference in which an implementation will be chosen by defining macros before including `secure_memzero.h`. If the developer does not express a preference, we choose the first available implementation in the order given above: platform-provided function if available, then memory barrier on GCC and Clang, then then volatile data pointer technique. Our defaults reflect what we believe are the best memory scrubbing approaches available today.

We have released our implementation into the public domain, allowing developers to use our function regardless of their own project license. We plan to keep our implementation updated to ensure it remains effective as compilers evolve. The current version of `secure_memzero.h` is available at

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

7 Scrubbing-Safe DSE

While we have tested our `secure_memzero` function with GCC, Clang, and Microsoft Visual C, by its very nature it cannot *guarantee* that a standard-conforming compiler will not remove our scrubbing operation. To address these cases, we implemented a scrubbing-safe dead store elimination option in Clang 3.9.0.

7.1 Inhibiting Scrubbing DSE

Our implementation works by identifying all stores that may be explicit scrubbing operations and preventing the dead store elimination pass from eliminating them. We consider a store, either a `store` IR instruction, or a call to LLVM's `memset` intrinsic, to be a potential scrubbing operation if

- ❖ 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 going be out of scope without being read.

The first two conditions are based on our observation how scrubbing operations are performed in the real code. The third allows a store that is overwritten by a later one to the same location before being read to be eliminated, which improves the performance. We note that our techniques preserves all dead stores satisfying the conditions above, regardless of whether the variables is considered sensitive or not. This may introduce false positives, dead stores to non-sensitive variables in memory that are preserved because they were considered potential scrubbing operations by our current implementation. We discuss the performance impact of our approach in Section 7.2.

It is worth considering an alternative approach to ensuring that sensitive data is scrubbed: The developer could explicitly annotate certain variables as *secret*, and have the compiler ensure that these variables are zeroed before going out of scope. This would automatically protect sensitive variables without requiring the developer to zero them explicitly. It would also eliminate potential false positives introduced by our approach, because only sensitive data would be scrubbed. Finally, it could also ensure that spilled registers containing sensitive data are zeroed, something our scrubbing-safe DSE approach does not do (see Section 8 for a discussion of this issue).

We chose our approach because it does not require *any* changes to the source code. Since developers are already aware of the need to clear memory, we rely on scrubbing operations already present in the code and simply ensure that they are not removed during optimization. Thus, our current approach is compatible with legacy code and can protect even projects that do not use a secure scrubbing function, provided the sensitive data is zeroed after use.

7.2 Performance

Dead store elimination is a compiler optimization intended to reduce code size and improve performance. By preserving certain dead stores, we are potentially preventing a useful optimization from improving the quality emitted code and improving performance. To determine whether or not this the case, we evaluated the performance of our code using the SPEC 2006 benchmark. We compiled and ran the SPEC 2006 benchmark under four compiler configurations: `-O2` only, `-O2` and `-fno-builtin-memset`, `-O2` with DSE disabled, and `-O2` with our scrubbing-safe DSE. In each case, we used Clang 3.9.0, modified to allow us to disable DSE completely or to selectively disable DSE as described above. Note that `-fno-builtin-memset` is *not* a reliable means of protecting scrubbing operations, as discussed in Section 3.2. The benchmark was run on a Ubuntu 16.04.1 server with an Intel Xeon Processor X3210 and 4GB memory.

Our results indicate that the performance if our scrubbing-safe DSE option is within 1% of the base case (`-O2` only). This difference is well within the variation of the benchmark; re-running the same tests yielded differences of the same order. Disabling DSE completely also did not affect performance by more than 1% over base in all but one case (`483.xalancbmk`) where it was within 2%. Finally, with the exception of the `403.gcc` benchmark, disabling built-in memset function also does not have a significant adverse effect on performance. For the `403.gcc` benchmark, the difference was within 5% of base.

8 Discussion

It is clear that, while the C standard tries to help by defining `memset_s`, in practice the C standard does not help. In particular, `memset_s` is defined in the optional Annex K, which is rarely implemented. Developers are then left on their own to implement versions of secure memset, and the most direct solution uses the volatile quantifier. But here again, the C standard does not help, because the corner cases of the C standard actually give the implementation a surprising amount of leeway in defining what constitutes a volatile access. As a result, any implementation of a secure memset based on the volatile qualifier is guarantee to work with every standard-compliant compiler.

Second, it's very tricky in practice to make sure that a secure scrubbing function works well. Because an incorrect implementation does not break any functionality, it cannot be caught by automatic regression tests. The only reliable way to test whether an implentation is correct or not is to manually check the generated binary, which can be time-consuming. What's worse, a seemingly working solution may turn out to be insecure under a different combination of platform, compiler and optimization level, which further increases the cost to test an implementation. In fact, as we showed in Section 5.2, developers did make mistakes in the implementing of secure scrubbing functions. This is why we implemented `secure_memzero` and tested it on Ubuntu, OpenBSD and Windows with GCC and Clang. We released it into the public domain so that developers can use it freely and collaborate to adapt it to future changes to the C standard, platforms or compilers.

Third, even if a well-implemented secure scrubbing function is available, developers will forget to use it, instead using the standard `memset` which is removed by the compiler. For example, we found this happened in Crypto++ (Section 5.9). This observation makes compiler-based solutions, for example the secure DSE, more attractive because they do not depend on developers correctly calling the right scrubbing function.

Finally, it's important to note that sensitive data may still remain in on the stack even after its primary storage location when it is passed as argument or spilled (in registers) onto the stack. Addressing this type of data leak requires more extensive support from the compiler.

9 Conclusion

Developers have known that compiler optimizations may remove scrubbing operations for some time. To combat this problem, many implementations of secure memset have been created. In this paper, we surveyed the existing solutions, analyzing the assumptions, advantages and disadvantages of them. Also, our case studies have shown that real world programs still have unscrubbed sensitive data, due to incorrect implementation of secure scrubbing function as well as from developers simply forgetting to use the secure scrubbing function. To solve the problem, we implemented the secure DSE, a compiler-based solution that keeps scrubbing operations while remove dead stores that have no security implications, and `secure_memzero`, a C implementation that have been tested on various platforms and with different compilers.

Acknowledgments

This work was funded in part by the National Science Foundation through grants NSF-1646493, NSF-1228967, and NSF-1237264.

References

[1] 6.45.2 Extended Asm - Assembler Instructions with C Expression Operands. `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`.

[2] #7419 (Choose a faster memwipe implementation) - Tor Bug Tracker & Wiki. `https://trac.torproject.org/projects/tor/ticket/7419`.

[3] 8537 – Optimizer Removes Code Necessary for Security. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537`.

[4] A glance at compiler internals: Keep my memset. `http://blog.quarkslab.com/a-glance-at-compiler-internals-keep-my-memset.html`.

[5] Bitcoin: Open source P2P money. `https://bitcoin.org/`.

[6] Bug 15495 - dead store pass ignores memory clobbering asm statement. `https://bugs.llvm.org/show_bug.cgi?id=15495`.

[7] Changeset 009521a. `https://community.openvpn.net/openvpn/changeset/009521ac8ae613084b23b9e3e5dc4ebeccd4c6c8/`.

[8] Crypto++ library. `https://www.cryptopp.com/`.

[9] CVE-2016-0777. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.

[10] GCC Bugzilla - Bug 26587. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=26587`.

[11] Kerberos - The Network Authentication Protocol. `https://web.mit.edu/kerberos/`.

[12] Libgcrypt. `https://www.gnu.org/software/libgcrypt/`.

[13] Libressl. `https://www.libressl.org/`.

[14] libsodium - A modern and easy-to-use crypto library. `https://github.com/jedisct1/libsodium`.

[15] The linux kernel archives. `https://www.kernel.org/`.

[16] musl libc. `https://www.musl-libc.org/`.

[17] Network Security Services - Mozilla. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS`.

[18] OpenSSH. `http://www.openssh.com/`.

[19] OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`.

[20] OpenVPN - Open Source VPN. `https://openvpn.net/`.

[21] Reimplement non-asm OPENSSL_cleanse(). `https://github.com/openssl/openssl/pull/455`.

[22] Sudo. `https://www.sudo.ws/`.

[23] Tarsnap - Online backups for the truly paranoid. `http://www.tarsnap.com/`.

[24] Tor Project: Anonymity Online. `https://www.torproject.org`.

[25] When erasing secrets, use a memset() that's not optimized away. `https://community.openvpn.net/openvpn/ticket/751`.

[26] WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud. `https://www.wolfssl.com`.

[27] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25, 2004.

[28] C. Deng and K. S. Namjoshi. Securing a compiler transformation. In *Proceedings of the 23rd Static Analysis Symposium*, SAS '16, pages 170–188, 2016.

[29] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops*, SPW '15, pages 73–87, 2015.

[30] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54, 2006.

[31] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 920–932, New York, NY, 2016.

[32] C. O'Donell and M. Sebor. Updated Field Experience With Annex K — Bounds Checking Interfaces. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm`, Sept. 2015.

[33] G. Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 2010.

[34] C. Percival. Erratum. `http://www.daemonology.net/blog/2014-09-05-erratum.html`.

[35] C. Percival. How to zero a buffer. `http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html`.

[36] R. Seacord. *The CERT C Secure Coding Standard*. Addison Wesley, 2009.

[37] C. C. Standard. Coding rules. `https://cryptocoding.net/index.php/Coding_rules#Clean_memory_of_secret_data`.

[38] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, 2013.

[39] D. Wheeler. Specially protect secrets (passwords and keys) in user memory. `https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/protect-secrets.html`.

A Appendix

A.1 Instrumenting Clang to Report DSE

To investigate how common it is for scrubbing operations to be removed by the compiler in open source projects, we developed a tool called *Scrubbing Finder*. Our case studies in Section 5 were performed with this tool.

Since scrubbing operations are removed in a compiler's dead store elimination optimization pass, we instrumented the DSE pass in LLVM/Clang 3.9.0 to report these instances. In order to differentiate removed scrubs from other dead stores, it is necessary to differentiate the different kinds of dead stores: (1) a store that is overwritten by another store with no read in between; (2) a store to an object that is about to be out of scope (a dead store to a stack object); (3) a store to an object that is about to be freed (a dead store to a heap object). There is no need to report the first case because even though the earlier store is indeed a scrubbing operation, it is safe to remove it. In addition, we noticed that all but one secure scrubbing implementation store a constant value to the buffer (typically zero). The only exception is the complicated computation technique of OpenSSL's `OPENSSL_cleanse` (see Section 3.4.1), which stores non-constants values—however, those stores are not dead stores. Thus the scrubbing finder only reports dead stores of (2) and (3) where a constant is stored.

Thus, when dead store belonging to one of the two categories described above is removed, *Scrubbing Finder* reports: (1) the *Location* of the removed scrub, including file and line number; (2) the *Removed IR Instruction*; and (3) *Additional Info* describing any instances where the removed scrub was inlined. Figure 9 is an example we found in Kerberos, which has since been fixed.

```
Location: src/lib/gssapi/krb5/lucid_context.c:269:13
Removed IR Instruction: call void @llvm.memset.p0i8.i64
  (i8* nonnull %call.i9.i, i8 0, i64 %conv.i8.i,
  i32 1, i1 false)
Additional Info:
  src/lib/gssapi/krb5/lucid_context.c:269:13 inlined at
  [src/lib/gssapi/krb5/lucid_context.c:285:13 inlined at
  [src/lib/gssapi/krb5/lucid_context.c:233:9 inlined at
  [src/lib/gssapi/krb5/lucid_context.c:94:16 ] ] ]
```

Figure 9: Example of a removed scrub in Kerberos reported by *Scrubbing Finder*.

In this example, the removed scrub is on line 269, column 13 of `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c`. Furthermore, the enclosing function of the removed operation is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:285:13`. The function containing line 285 of `lucid_context.c` is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:233:9`. The function containing line 233 of `lucid_context.c` is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:94:16`.