Sanitizing Sensitive Data: How to get it Right (or at least Less Wrong…)

Roderick Chapman

Protean Code Limited, UK
rod@proteancode.com

Abstract. Coding standards and guidance for secure programming call for sensitive data to be "sanitized" before being de-allocated. This paper considers what this really means in technical terms, why it is actually rather difficult to achieve, and how such a requirement can be realistically implemented and verified, concentrating of the facilities offered by Ada and SPARK. The paper closes with a proposed policy and coding standard that can be applied and adapted to other projects.

Keywords: Security, Sanitization, SPARK, Verification, Volatile, Optimization, Proof.

1 The Problem

Secure systems must be built to resist attack by increasingly sophisticated adversaries. An attacker might be able to observe or provoke a system into "leaking" or revealing data that is supposed to be kept secret, such as cryptographic keys, supposedly random and unique "nonce" values, the plaintext of passwords and so on. A well-documented example is the "Heartbleed" vulnerability in the OpenSSL libraries that allowed a buffer over-read attack to leak sensitive data that had been left unsanitized in memory.

Several coding standards and "guidance" documents exist that call for "sensitive" data to be "sanitized" when no longer needed, but offer little advice on how this is to be achieved or verified, especially given the complexity of modern programming languages and hardware. This paper considers this problem in more detail and describes the key technical challenges, before going on to consider the facilities offered by Ada and SPARK that can be used to meet these demands, based on experience gained from a recent project.

1.1 Why is sanitizing data hard?

Sanitizing sensitive data might seem simple at first: just overwrite the data with zeros and carry on, right? A less trivial analysis, though, reveals several important questions, including:

- How do we define “sensitive”? What objects in the program are “sensitive” and how are they identified?
- Imagine that we have two variables A and B which are defined to be “sensitive.” We declare and initialize a local variable C with an initial value derived from some function that combines A and B. Is C “sensitive”? Does C need to be sanitized?
- Can constant objects be sensitive? If so, how are they to be sanitized?
- Exactly *when* should sanitization be performed? This question is related to the scope and lifetime of data objects which is, in turn, intricately entwined with a particular programming language’s model of how data should be organized and (de-)allocated.
- Compiler optimization might remove a sanitizing assignment to an object if the assignment is seen to be redundant or “dead” by the optimizer. How can this be prevented?
- How do we verify that sanitization really has been performed correctly, to the satisfaction of ourselves, our customers, and regulators?

1.2 Standards, Guidance and Problems

There are several (possibly far too many) sets of guidance or coding rules for secure systems that call for sensitive data to be sanitized as soon as it is no longer needed, so that (for example) a subsequent buffer over-read will not find any useful data. This section considers an incomplete set of these, and tries to point out problems in meeting their advice.

CESG
The UK’s national technical authority for secure software, CESG, offers a short (but thankfully unclassified) “Guidance Note” on secure coding [1]. It offers some generic advice, but mainly consists of coding rules for C and C++. A need to avoid copying sensitive data is mentioned (to avoid a copy existing even if the original is sanitized), with two paragraphs specifically on sanitization:

“Sanitise all variables that contain sensitive data (such as cryptovariables and unencrypted data) by overwriting with zeroes once they are no longer needed. This includes all copies of the data: call-by-value functions (as found in C) implicitly copy the value of their parameters, so their parameters should always be sanitised before the function exits. At protective markings higher than Restricted, sanitisation may require multiple overwrites or verification, or both.” [1, para 58].

and

“The sanitisation is needed because errors may result in the disclosure of a block of memory, therefore the risk of that memory containing anything useful needs to be minimised. The size of the data is not a factor: even single bytes need to be sanitised, since in some cases a difference of 8 bits could have a significant impact on the practicality of an attack. On the other hand, the lifetime of data may be a factor: if a variable can be shown to be overwritten shortly afterwards, it may be acceptable not to sanitise it, provided it is sanitised when it is no longer needed. 'Shortly' is not defined more precisely, since it will depend on the situation…" [1, para 59]

This is well-meaning, but offers little in the way of real technical detail of how sanitization is to be achieved or verified. The failure to define "shortly afterwards" is also disappointing.

CERT Coding Standards

The US-led CERT has produced coding guidance for secure software development, covering C, C++, Java and Perl to date, with several tool vendors claiming compliance. The CERT C Coding Standard provides some advice on sanitization:

- Recommendation 08 (Memory Management), Item 06 is titled "Ensure that sensitive data is not written out to disk" which mostly covers the problem of an operating system "paging out" sensitive data to a disk or an application doing a "core dump" which writes the state of a process to a disk file, potentially revealing the state of sensitive data. These are valid concerns, relevant to any application running on an operating system that supports paging and so on, so not really a "C language issue" per-se, since these problems could affect code written in any language.
- Recommendation 48 (Miscellaneous), Item 06 is titled "Beware of compiler optimizations" and covers the problem of a compiler removing a sanitizing assignment. It goes on to recommend using "optimization safe" C functions, C's "volatile" qualifier (more of which later…) or operating-system specific functions that are designed to sanitize memory.

Both of these recommendations appear to presume the existence of some sort of operating system (and possibly a "disk"), but what if we're programming an embedded "bare metal" system with no OS at all? How can we sanitize data properly in such an environment?

Cryptography Coding Standard

The Cryptography Coding Standard is "a set of coding rules to prevent the most common weaknesses in software cryptographic implementations" [3]. Their coding rules touch on sanitization in a number of places:

- Coding Rule 5 "Prevent compiler interference with security critical operations" mentions the problem of compilers removing sanitizing assignments, and how even a call to C's standard "memset" function can be optimized away in some cases. It offers the rather vague advice to "Look at the assembly code produced and check that all instructions are there" which hardly seems practical for anything but trivial code. It also recommends "consider disabling compiler optimizations that can eliminate or weaken security checks" but again this seems impractical – modern compilers have *hundreds* of optimization switches, which makes it almost impossible to "know" which set of them will or won't "interfere" with security. Finally, rule 5 does point out that the 2011 C standard does include a new "memset_s" function, a call to which is explicitly *not* allowed to be optimized.

- Coding Rule 11 "Clean memory of secret data" looks promising, recommending that code should "Clear all variables containing secret data before they go out of scope." It points out the existence of a SecureZeroMemory function in the Win32 API for this purpose. It also offers a portable C function that can be used to overwrite memory that works "for non-buggy compilers" [sic].

1.3 Technical Issues

Having seen that the standards and guidance documents offer well-meaning but imprecise advice, we now turn to a selection of more detailed technical problems.

Unwanted Compiler Optimization
Several of the guidance documents cited above refer to this problem, so it warrants more attention here.

Modern implementations of computer architectures feature a marked difference between the access time of CPU registers, data cache(s), and main memory, sometimes by many orders of magnitude. In short, DRAM access times have not kept pace with CPU clock rates, so the penalty for a "register miss" or a "cache miss" is pronounced. Modern compilers therefore devote significant effort in several, related classes of optimization, including:

1. Common sub-expression elimination and partial redundancy elimination. These prevent semantically equivalent expressions from being evaluated more than once.
2. Register allocation and tracking, so that variables and the values of expressions are stored in CPU registers as much as possible.
3. Dead-load and dead-store elimination.

These improve average-case performance, but create some issues for sanitization:

- Guidance calls for the "memory" occupied by a sensitive variable to be overwritten "before the variable goes out of scope", but what does that mean if the variable only ever exists in an internal CPU register and there is no "memory" allocated for it at all?
- A final sanitizing assignment needs to occur just before a variable "goes out of scope", so is (by definition) a "dead store" in the eyes of an optimizer, so might be removed, on the assumption that once a variable has gone out of scope it can't be accessed any more. This creates a conflict in the compiler: we (the programmers) want dead stores to be retained for one or more particular variables, but the compiler is trying its hardest to remove them in the interests of improving performance of the generated code.

Derived values and copies
In his thorough analysis "Zeroing buffers is insufficient" [4], Percival points out several more pernicious technical issues with a simple "write zeros into memory" approach. Specifically, he points out:

- Sanitizing the *one* memory block where a variable is stored is not good enough. Compilers implicitly make copies of data into registers or implicitly-declared and initialized local variables, so these might also contain a copy of some sensitive information that needs to be sanitized. In the worst case, a compiler might evaluate the value of a sensitive variable into a CPU register *and* spill that register into an implicitly allocated temporary variable on the stack. There is no way to portably sanitize such temporary variables in C or Ada, since those variables do not appear in the source code.
- If a sensitive piece of data is left in a CPU register, you *cannot* assume that that CPU register will be re-used and the data over-written "quickly". Percival points out that some CPU registers (such as the SSE registers on x86) are rarely used, and some registers are specifically designed for cryptographic algorithms such as AES – the problem being that you carefully use a "special" register to hold an AES key (for example), but then that register is not used for anything else in your program, so the key value just sits there for ages and is never overwritten.

A related problem is that of derived values. As pointed out in section 1.1, if two sensitive variables A and B are combined in some way to get a value in variable C, should C be considered to be sensitive and therefore needing sanitization? The answer is "it depends"… on the exact operation used to derive C, the nature of A and B, and so on. It is far from simple to suggest a generic one-size-fits-all policy for such variables.

By-copy parameter passing
If a subprogram parameter is passed by copy, then the value of the actual parameter is copied into the storage associated with the formal parameter (which might be "memory", or might be a CPU register.) If the actual is sensitive, then so is the formal parameter. In Ada, this is particularly problematic, since "in" mode parameters are constant and so cannot be assigned to at all, and the choice between by-copy and by-reference passing can be unspecified for some types.

CPU data caching and memory hierarchy
Anyone that has programmed a device-driver on a "bare metal" target will know that the presence of a "write" instruction does *not* guarantee that the data actually reaches the target hardware device at all, or in the order indicated in the source code. Modern CPUs have multiple levels of data caching, which may be in "write back" mode, so an instruction to write a particular word of memory might not actually reach the main memory device until the offending data cache line is flushed or invalidated. Secondly, modern CPUs can re-order memory accesses in rather unexpected ways, which can complicate matters further.

Some operating systems offer functions that are specifically designed to securely sanitize memory, such as Win32's SecureZeroMemory function. We presume these functions take care of any required flushing of caches, paged-out data and so on.

On bare-metal targets, we might turn off all data caching or insist on "write through" mode, but this may be Draconian, since disabling all caching for all stack-allocated data would incur a potentially huge performance penalty. Some CPUs might allow special instructions to flush particular cache lines and instruct the CPU to pause until all queued memory accessed are complete. These techniques are valid (and indeed may be absolutely necessary), but require recourse to obviously non-portable assembly language programming at some level.

1.4 An example – how it can go wrong in Ada

This section closes with a short (and somewhat contrived) example of how sanitization can fail in Ada. In the remainder of the paper, all examples have been compiled with the GPL 2016 Edition of GNAT for 32-bit x86 running on Windows 7 Pro.

Consider a simple procedure GK that takes three seed values A, B, and C, and produces a derived key value K from them. For example:

```
subtype Word32 is Interfaces.Unsigned_32;
procedure GK (A, B, C : in     Word32;
              K       :    out Word32);
```

The body of GK combines A, B, and C using a local, temporary variable T which we have decided is sensitive and needs to be sanitized with a final assignment, thus:

```
procedure GK
   (A, B, C : in     Word32;
    K       :    out Word32)
is
   T : Word32;
begin
   T := A xor B; -- line 15
   T := T xor C;
   K := T;

   -- Now sanitize T
   T := 0; -- line 20
end GK;
```

To see what's going on, we'll compile with both "-g" and "-fverbose-asm" flags. We'll also enable all warnings with "-gnatwa" and "-Wall" as we would on any real project. Compiling GK does yield a warning:

```
p1.adb:20:07: warning: useless assignment to "T", value never
referenced
```

which hints at trouble ahead. Compiling with –O0 (little or no optimization) yields the following assembly language for lines 15 through 20 of GK:

```
.loc 1 15 0
movl   8(%ebp), %eax    # a, tmp88
xorl   12(%ebp), %eax   # b, tmp87
movl   %eax, -12(%ebp)  # tmp87, t
.loc 1 16 0
movl   16(%ebp), %eax   # c, tmp89
xorl   %eax, -12(%ebp)  # tmp89, t
.loc 1 17 0
movl   -12(%ebp), %eax  # t, tmp90
movl   %eax, -16(%ebp)  # tmp90, k
.loc 1 20 0
movl   $0, -12(%ebp)    #, t
```

so we can see the final assignment to T on line 20 has indeed been generated as a single "movl" instruction.

Turning on the optimizer at level "-O1" reveals a different story. For the same fragment of code, we get:

```
movl   16(%ebp), %eax   # c, c
xorl   12(%ebp), %eax   # b, D.3010
xorl    8(%ebp), %eax   # a, k
```

and that's all. The local variable T is not allocated at all on the stack – it has completely disappeared, in fact, with the intermediate results left in the CPU register EAX. Our well-intended attempt to sanitize T has been discarded by the compiler, but then again, T has disappeared entirely, so is this sufficient? What about the intermediate value left in EAX – is that overwritten "soon" by the calling subprogram perhaps?

2 Sanitization – Constraints and Goals

In developing the coding standard for a recent project, we had to meet both CESG's guidance for sanitization [1], but also the constraints imposed by the wider demands of the project, including the runtime environment, compilers, feature of the target platform and its operating system and so on.

In searching for the most general solution, we tried to respect the following constraints:

1. The approach to sanitization should minimize dependence on predefined library units and the use of language features that require substantial support from the Ada runtime library. In particular, for our project, we required compatibility with GNAT's "Zero Footprint" (ZFP) runtime library.
2. The approach should not depend on any operating system facilities, and so can be deployed on a "bare metal" target system.

3. The approach should be compatible with the SPARK language (either SPARK2005 or SPARK2014) and verification tools.

Secondly, what does a "good" approach to sanitization look like? In developing these guidelines for Ada, we tried to respect the following goals:

1. Any proposed approach should be *portable* in that it should not depend on non-standard behavior from the compiler, and should not rely on particular *unspecified* or *implementation-defined* choices made by a compiler.
2. Our approach should permit optimization to be enabled at all levels, with sufficient confidence that sanitization code would be preserved and implemented correctly.
3. Our approach must prevent (as far as is possible) explicit or implicit copying or assignment of sensitive values. This also affects parameter passing, since a "by-copy" formal parameter involves assignment.
4. Our approach should facilitate (or at least not obstruct) verification with the SPARK toolset.
5. Our approach should meet or exceed the demands of the various regulatory standards, such as [1]. Furthermore, we should be able to explain and justify our approach to those regulators so that we can convince them that it actually works.

3 Sanitization mechanisms in Ada

Having considered the scope of this problem, this section turns to the language-based mechanisms that are available in Ada. Knowing what mechanisms are available can then lead to a policy that can be adopted for a particular project.

3.1 Volatile

Ada, C and C++ all include a facility to mark an object as "Volatile", meaning that the compiler must respect the exact sequence of reads and writes to such an object that are indicated in the source code. Ada goes further, allowing Volatile *types* as well as objects. The Ada LRM offers a clear implementation requirement (Ada2012 RM, C.6(20)):

"The external effect of a program…is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program."

Let's see what happens to our example procedure GK with the declaration of T changed as follows:

```
T : Word32 with Volatile;
```

With that in place, we should be able to turn the optimizer "up to 11" (well…3) and compile with "-O3". Firstly, the warning from the front-end about the useless assignment to T disappears, which is a good sign. The generated code for lines 15 – 20 is:

```
.loc 1 15 0
movl   12(%ebp), %eax   # b, b
xorl    8(%ebp), %eax   # a, D.3014
movl   %eax, -12(%ebp)  # D.3014, t
.loc 1 16 0
movl   -12(%ebp), %eax  # t, D.3015
xorl    16(%ebp), %eax  # c, D.3014
movl   %eax, -12(%ebp)  # D.3014, t
.loc 1 17 0
movl   -12(%ebp), %eax  # t, k
.loc 1 20 0
movl   $0, -12(%ebp)  #, t
```

so we see that *all* the reads and writes of T have been preserved, including the final sanitizing assignment.

At first glance, this appears to be a perfect match, at least when it comes to preventing the optimization of sanitizing assignments. Unfortunately, it's not that simple for several reasons:

1. Volatile prevents optimization of *all* reads and writes to an object, but we only require that the *final* sanitizing assignment is preserved, so use of Volatile might have a serious but unnecessary impact on the performance of the generated code.
2. SPARK2014 places some restrictions on the use of Volatile that might clash.
3. Most seriously and worryingly, Regehr and Eide [5] have shown that most, if not all, compilers can mis-compile Volatile and *do* optimize away reads and writes when they shouldn't. Regehr's work dates from 2008 (so we hope compilers have improved since then), and was based on analysis of C programs, but his concerns are real, especially since his results include those for GCC, which shares its back-end (and optimization code) with GNAT.

So, despite its initial good looks, the use of Volatile is not a panacea for data sanitization. Secondly, it does not address the need to restrict assignment and copying of sensitive data objects at all.

3.2 Controlled Types

Ada's "Controlled Types" offer a tempting approach to supply a "Finalize" procedure that sanitizes an object. Unfortunately, the use of controlled types conflicts with our constraints to be compatible with both SPARK and the ZFP runtime library. They are also renowned for their complexity, so were rejected without further investigation.

3.3 Limited Types

Ada's limited types are particularly attractive for holding sensitive data. Firstly, the programmer can have complete control over exactly what set of operations are available to clients. Secondly, and by default, assignment is not defined for limited types.

Finally, an explicitly limited record type is defined to be a *by-reference* type (RM 6.2(7)) so we can be sure that all formal parameters of such a type will be passed by reference, not by copy.

3.4 By-Reference Types

Where the use of a limited record type is not appropriate or practical, there are still other means of forcing a type to be a “by-reference” type in Ada, which will, at least, prevent copying by parameter passing where we don’t want it. LRM C.6 (18) tells us that if any sub-component of a type is Atomic or Volatile, then the type is defined to be a by-reference type. Thus we can force by-reference passing for even a simple scalar type by wrapping it in a record which has a single Atomic or component. For example, instead of declaring a formal “in” mode parameter of type Boolean, we might declare:

```
type Sensitive_Boolean is record
   F : Boolean with Volatile;
end record;
```

to ensure by-reference parameter passing. Using GNAT, it is also possible to *verify* the parameter passing mechanism using the “–gnatRm” flag.

3.5 Pragma Inspection_Point

This little-used (and little-understood perhaps) pragma has particular relevance to this problem. Inspection_Point was introduced in Ada95 as part of the RM’s Safety and Security Annex H. It is designed to specify a list of objects that must be *inspectable* at a particular point in a program. A pragmatic interpretation means that the listed objects are supposed to be stored in memory at the inspection point so that their values can be seen by external means, such as a logic analyser, a JTAG probe, a real-time debugger or similar. From the point of view of optimization, the Ada RM is clear:

‘The implementation is not allowed to perform “dead store elimination” on the last assignment to a variable prior to a point where the variable is inspectable. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.’ (Ada RM H3.2 (9)).

This seems ideal for our needs – if a final, sanitizing assignment to a sensitive object is immediately followed by a pragma Inspection_Point for that object, then that final assignment should not be optimized away. This provides much finer control than pragma Volatile. For the curious, GNAT actually implements pragma Inspection_Point by generating a dummy volatile read to each of the objects specified in the pragma. See the file gcc-interface/trans.c in the GNAT sources for details [6] and search the file for “Inspection_Point”.

Returning to our simple example, we revert to declaring T as a normal (non-volatile) local variable, but now follow the final assignment with an Inspection_Point, thus:

```
-- Now sanitize T
T := 0; -- line 20
pragma Inspection_Point (T);
```

The generated code at –O1 is:

```
/APP
 # 21 "p3.adb" 1
 # inspection point: t is in $0   #
 # 0 "" 2
/NO_APP
movl  12(%ebp), %eax  # b, b
xorl  16(%ebp), %eax  # c, D.3010
.loc 1 16 0
xorl  8(%ebp), %eax   # a, k
```

which is interesting. Again, the variable T has been entirely eliminated, but commentary has been added that "t is in $0" since T does not have an accessible address in memory at all.

3.6 No_Inline and sanitizing operations

Having identified the problems with Volatile objects, Regehr and Eide go on to recommend that all reads and writes of a volatile variable should be performed by a subprogram call that can never be inlined, since inlined code has the potential to be optimized away during the compilation of any calling units. Regehr demonstrates how this works well for C, and the equivalent mechanism exists for Ada with the GNAT-defined pragma No_Inline.

Combining this idea with the use of a limited private type for sensitive data yields the following pattern for a sensitive abstract data type:

```
package Sensitive is
   type T is limited private; -- so no assignment

   procedure Sanitize (X : out T);
   pragma No_Inline (Sanitize);
private
   type T is limited record -- so by-reference
       F : … -- whatever…
   end record;
end Sensitive;
```

The body of Sensitive.Sanitize might depend on the target platform and operating system, so we recommend implementing it as a separate subunit of package Sensitive to allow for alternative implementations to be chosen at build-time. Let's imagine that the field F of type T is of type Word32. In that case, a suitable implementation for a bare-metal/ZFP target might be:

```
separate (Sensitive)
procedure Sanitize (X : out T) is
begin
   X.F := 0;
   pragma Inspection_Point (X);
end Sanitize;
```

At –O3, the generated code for the assignment statement and the pragma is:

```
movl   8(%ebp), %eax # x, x
movl   $0, (%eax)  #, x_2(D)->f
/APP
 # 6 "sensitive-sanitize.adb" 1
 # inspection point: x address is in %eax # x
 # 0 "" 2
/NO_APP
```

We can also check the parameter passing mechanism using –gnatRm which yields:

```
procedure sanitize declared at sensitive.ads:5:14
  convention : Ada
  x : passed by reference
```

4 Verification and SPARK

The SPARK toolset offers two major forms of static verification—information-flow analysis and proof of user-defined assertions, pre-conditions and post-conditions. This section briefly considers the interplay between sanitization and these forms of verification.

4.1 Information Flow Analysis

As expected, both the SPARK2005 and SPARK2014 tools will reliably report that a final sanitizing assignment to a local variable is *ineffective*, meaning that the assignment has no influence on the final value of any exported variable of the subprogram under analysis. This is perfectly correct and reasonable. At first, such errors being reported might seem an annoyance, we can turn this to our advantage using pragma Warnings to document the expectation and need for the sanitization. For our earlier example, we would add:

```
pragma Warnings (Off, "unused assignment",
                 Reason => "Sanitization");
T := 0;
pragma Inspection_Point (T);
```

4.2 Proof

At first glance, it might be possible to prove that sanitization of variables has been performed, but closer inspection reveals two main issues:

- The final value of a *local* variable cannot be asserted in the post-condition of the subprogram that declares it, owing to its very local-ness. It would be possible to assert the value of a library-level variable (which would appear in the *Global* aspect of the subprogram).
- Writing an assertion regarding the *value* of a sensitive variable means that we need to decide on a (constant) value that should be used to sanitize the variable. The naive approach of "zero all bits" might not be appropriate, since "all zeros" might not be a valid value. SPARK and Ada have no "memset" or similar, so need to be able to write an assignment statement which is legal and itself free from runtime errors.

5 A Policy for Sanitization

In light of the difficulties described above, and the facilities offered by Ada and SPARK, and our experience on one project, we would offer the following policy for sanitization of sensitive data for future work.

5.1 Identification and Naming of Sensitive Variables

A project must document a clear policy for what exactly is and isn't considered to be a "sensitive" object. This is clearly project- and application-specific. In cryptographic applications, for example, sensitive data might include cryptographic keys, the primitive seed values used to generate such keys, use-one random "nonce" values, initialization vectors for encryption algorithms, and "trust anchors" or private keys.

The definition of "sensitive" may also have to consider the visibility and lifetime of the objects—local variables and library level states might have to be treated very differently, for example.

Finally, the sensitivity of a particular variable might depend on the precise physical storage device it is allocated to. For example, general-purpose DRAM might be considered "insecure" since its operation could be observed by a sophisticated attacker, but the private RAM on a tamper-proof hardware security module might be considered suitable for storage of sensitive information with no sanitization required.

Having chosen a policy for deciding which states are sensitive, we propose a naming convention as follows:

- The names of *types* used for sensitive data should be prefixed with "Sensitive_".

- The names of *variables* that are sensitive should have the suffix “_SAN” meaning that such variables should be sanitized.
- The name of a formal subprogram parameter that *might* be associated with a sensitive actual parameter shall also have the suffix “_SAN”.
- Sensitive *constants* are not permitted.

5.2 Types and Aspects for Sensitive Data

- By-reference types should be used for all sensitive data.
- Preferably, and if possible, a limited type should be used for sensitive data to forbid assignment. In this case:
  - A “Sanitize” procedure should be supplied, as shown in section 3.6, which has a No_Inline pragma applied to it.
  - The body of such a “Sanitize” procedure should be supplied as a separate subunit to allow for multiple implementations for different platforms and operating systems.
  - The body of “Sanitize” shall include a pragma Inspection_Point immediately following the final assignment to the formal parameter. Note that the presence of the pragma is sufficient to suppress the “useless assignment” warning illustrated in section 1.4. This is useful for verification, since presence of this warning is a strong indication that the programmer has forgotten to add the pragma.
- If a limited type is not possible, then a Sanitize procedure shall still be supplied for any particular sensitive type, implemented as above. In this case, code review checklists must include a check that assignment is not used for objects of such types.
- For SPARK code, a pragma Warnings shall always precede a final sanitizing assignment (or the call to a Sanitize procedure) to document the need for the sanitization and to suppress the information-flow warning.

5.3 Compiler Switches and Analysis

- All code should be compiled with “-gnatwa” to ensure that the “useless assignment” warning is generated. This should be expected for sanitizing assignment, but suppressed with pragma Inspection_Point.
- The “-gnatRm” switch should be used to verify by-reference parameter passing mechanism for all sensitive formal parameters. This is easy is the naming convention above has also been followed.
- For the utmost confidence, analysis of the generated assembly language should be performed using the “-g” and “-fverbose-asm” flags to verify that the inspection points required are present and correct.

6 Future Work

Several areas of potential future work seem apparent.

- Several authors have called for compilers to help automate sanitization via some sort of special compilation switch ("-fsanitize_local_data" perhaps?). This could go further than source-based techniques since the compiler could arrange to sanitize *all* local states, derived variables, temporaries, and CPU registers for example. How a compiler designer would convince others of the correctness of such an approach remains unknown.
- The problem of sensitive derived variables could be addressed through more advanced information flow analysis. If a tool like GNATProve, for example, knew that variables A and B were sensitive, then could it automatically infer that C (derived from A and B) were also sensitive?
- At the Ada language level, is there a need for a new aspect that could be applied to types or variables to mark them as requiring sanitization? This would provide a finer level of control than a simple, global compiler switch.

7 Conclusions

Sanitization of sensitive data remains a thorny issue: standards call for it to be done, but offer little advice on how it should be achieved in practice or verified. This paper has illustrated some of the problems and shown how they can be addressed in Ada and SPARK and developed into a policy, coding standard, and verification strategy for a particular project.

Acknowledgements. The author would like to Rod White and Bill Ellis for their comments on an early draft of this paper.

8 References

1. CESG. Coding Requirements and Guidance (IA Developers' Note 6), CESG, Issue 1.1, October 2015. https://www.ncsc.gov.uk/guidance/coding-requirements-and-guidance-ia-developers-note-6
2. US CERT. SEI CERT C Coding Standard. https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
3. Cryptography Coding Standard Project https://cryptocoding.net/index.php/Cryptography_Coding_Standard
4. Percival, C.: Zeroing Buffers is Insufficient. http://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html
5. Regehr, J., Eide, E.: Volatiles are Miscompiled and What to Do About It. In: Proceedings of the Eighth ACM and IEEE International Conference on Embedded Software (EMSOFT), Atlanta, Georgia, October 2008. http://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf
6. GNAT sources at gcc.gnu.org. https://gcc.gnu.org/viewcvs/gcc/trunk/gcc/ada/gcc-interface/trans.c