

Tentative Definition of Secret Attribute in GCC



Pierrick Philippe
pierrick.philippe@irisa.fr

FOSDEM'26

31/01/2026

1. Introduction

GCC Static Analyzer (*a.k.a.* GSA)

- Introduced in 2020 in GCC 10.
- Operates on GIMPLE SSA after most of the optimizations:
 - optimization-dependent analysis.
- Framework for security-related analysis:
 - mostly memory-related.

Null-Pointer Dereference

```
1 int foo(int *arg) {  
2     return *arg;  
3 }  
4  
5 int main(void) {  
6     int *ptr = NULL;  
7     int *aliasing = ptr; // aliasing ptr  
8  
9     return foo (aliasing);  
10 }
```

Listing 1.2: A  code snippet with an interprocedural NPD.

Null-Pointer Dereference

gcc -fanalyzer test.c

```
./test.c: In function 'foo':
./test.c:3:10: warning: dereference of NULL 'arg' [CWE-476] [-Wanalyzer-null-dereference]
 3 |     return *arg;
   |           ^~~~
'main': events 1-4
 6 | int main(void) {
   |     ^~~~
   |     (1) entry to 'main'
 7 |     int *ptr = NULL;
   |           ~~~
   |           (2) 'ptr' is NULL
 8 |     int *aliasing = ptr; // aliasing ptr
   |           ~~~~~~
   |           (3) 'ptr' is NULL
 9 |
10 |     return foo (aliasing);
   |           ~~~~~~
   |           (4) calling 'foo' from 'main'
+--> 'foo': events 5-6
 2 | int foo(int *arg) {
   |     ^~~
   |     (5) entry to 'foo'
 3 |     return *arg;
   |           ~~~
   |           (6) dereference of NULL 'arg'
```

Secret-related Example

```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1;
11    }
12    return res;
13 }
```



Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example

```
1 int main(void) {  
2     char password[256];  
3     char *expected = "fubar";  
4     int res;  
5     get_password (password);  
6     if (validate (password, expected))  
7         res = 0;  
8     else {  
9         printf ("Wrong password, expected password was: %s)", expected);  
10        res = 1;  
11    }  
12    return res;  
13 }
```



Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example

```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1;
11    }
12    return res;
13 }
```



Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example

```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1;
11    }
12    return res;
13 }
```



Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example

```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1;
11    }
12    return res;
13 }
```



Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example

```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1;
11    }
12    return res;
13 }
```




Listing 1.3: A vulnerable and naive authentication  code snippet.

Secret-related Example


```
1 int main(void) {
2     char password[256];
3     char *expected = "fubar";
4     int res;
5     get_password (password);
6     if (validate (password, expected))
7         res = 0;
8     else {
9         printf ("Wrong password, expected password was: %s)", expected);
10        res = 1; ← Not detected by the GSA!
11    }
12    return res;
13 }
```

Listing 1.3: A vulnerable and naive authentication  code snippet.

How Is It Possible?

- A *semantic gap* remains between programming language and secret-leakage.
 - specifically for 

How Is It Possible?

- A *semantic gap* remains between programming language and secret-leakage.
 - specifically for 


Functional property

Describes what a program should do (e.g., not crash).

Non-functional property

Describes how a program does what it does (e.g., its safety or security).

How Is It Possible?

- A *semantic gap* remains between programming language and secret-leakage.
 - specifically for 

Functional property


Describes what a program should do (e.g., not crash).

← NPD falls here

Non-functional property

Describes how a program does what it does (e.g., its safety or security).

How Is It Possible?

- A *semantic gap* remains between programming language and secret-leakage.
 - specifically for 

Functional property

Describes what a program should do (e.g., not crash).

← NPD falls here

Non-functional property

Describes how a program does what it does (e.g., its safety or security).

↙ Printing a password falls here

 has no concept of **secret** variables.

Secret-related Analysis Ecosystem

- Lots of tools exists:
 - usually tied to **specific vulnerabilities**
- Adoption remains marginal (Johnson et *al.*^[1], Sadowski et *al.*^[2]):
 - break **developer workflows**
 - have **complex or noisy** diagnostics

^[1] “*Why don’t software developers use static analysis tools to find bugs?*”, Johnson et *al.*, ICSE, 2013.

^[2] “*Lessons from building static analysis tools at Google*”, Sadowski et *al.*, ACM Communications, 2018.

Secret-related Analysis Ecosystem

- Lots of tools exists:
 - usually tied to **specific vulnerabilities**
- Adoption remains marginal (Johnson et *al.*^[1], Sadowski et *al.*^[2]):
 - break **developer workflows**
 - have **complex or noisy** diagnostics } Usability issues

^[1] “*Why don’t software developers use static analysis tools to find bugs?*”, Johnson et *al.*, ICSE, 2013.

^[2] “*Lessons from building static analysis tools at Google*”, Sadowski et *al.*, ACM Communications, 2018.

Secret-related Analysis Ecosystem

- Lots of tools exists:
 - usually tied to **specific vulnerabilities**
- Adoption remains marginal (Johnson et al.^[1], Sadowski et al.^[2]):
 - break **developer workflows**
 - have **complex or noisy** diagnostics} Usability issues
- Aiming for:
 - usability:
 - ⇒ GCC is **already** part of developer's workflow
 - ⇒ The GSA **already** implements a reporting system
 - and **soundy** analysis

[1] “Why don't software developers use static analysis tools to find bugs?”, Johnson et al., ICSE, 2013.

[2] “Lessons from building static analysis tools at Google”, Sadowski et al., ACM Communications, 2018.

Could we detect **secret-dependent** data being passed to **leaking** functions with the GSA?

Secret-dependency

Secrets might propagate to other variables during execution.

```
1 void foo (void) {  
2     int secret = get_secret ();  
3     int x = secret + 42;  
4     secret = 0; // zeroization  
5 } // x depends on secret and remains in memory
```



Listing 1.1: A simple  code snippet to illustrate secret-dependency.

2. A Secret Attributes System

From Binary Model To Language Model

Concept	Programming language	Binary
Variable	Enriched with meta-information : <ul style="list-style-type: none">• specifiers (e.g., <code>const</code>),• scope (e.g., local or global),• type (e.g., <code>int</code> or <code>struct foo</code>)	Does not exist outside constants. Only addresses and registers.

Table 2.1: Programming languages have higher concepts than binaries.

From Binary Model To Language Model

Concept	Programming language	Binary
Variable	Enriched with meta-information : <ul style="list-style-type: none">• specifiers (e.g., <code>const</code>),• scope (e.g., local or global),• type (e.g., <code>int</code> or <code>struct foo</code>)	Does not exist outside constants. Only addresses and registers.
Function	Defined or only declared: <ul style="list-style-type: none">• may or not have input(s) (i.e., parameter(s))• may or not have output(s)	Does not really exist, only <i>labels</i> . <i>Symbols</i> are specific <i>labels</i> , with context setup before jumping (e.g., <code>call</code> instruction).

Table 2.1: Programming languages have higher concepts than binaries.

Designing Secret Attribute In GCC

```
1 int my_function (void) {
2     /* "passwd" is a local variable and becomes secret on its definition */
3     char *passwd = get_user_input ();
4     validate (passwd);
5 }
6
7 struct RSA_keys {
8     char public_key[256];
9     /* "private_key" field should be secret for all instances */
10    char private_key[256];
11 };
12
13 /* "key" will always be secret when the function begins */
14 int encrypt (char *key, char *buf) { ... }
15
16 /* "output_key" becomes secret when the function returns */
17 int generate_secret_key (char *output_key);
```



Listing 2.1: Different ways of creating secrets in .

Designing Secret Attribute In GCC

```
1 int my_function (void) {
2     /* "passwd" is a local variable and becomes secret on its definition */
3     char *passwd = get_user_input (); ← Local variable
4     validate (passwd);
5 }
6
7 struct RSA_keys {
8     char public_key[256];
9     /* "private_key" field should be secret for all instances */
10    char private_key[256];
11 };
12
13 /* "key" will always be secret when the function begins */
14 int encrypt (char *key, char *buf) { ... }
15
16 /* "output_key" becomes secret when the function returns */
17 int generate_secret_key (char *output_key);
```



Listing 2.1: Different ways of creating secrets in .

Designing Secret Attribute In GCC

```
1 int my_function (void) {
2     /* "passwd" is a local variable and becomes secret on its definition */
3     char *passwd = get_user_input (); ← Local variable
4     validate (passwd);
5 }
6
7 struct RSA_keys {
8     char public_key[256];
9     /* "private_key" field should be secret for all instances */
10    char private_key[256]; ← Type definition
11 };
12
13 /* "key" will always be secret when the function begins */
14 int encrypt (char *key, char *buf) { ... }
15
16 /* "output_key" becomes secret when the function returns */
17 int generate_secret_key (char *output_key);
```


Listing 2.1: Different ways of creating secrets in .

Designing Secret Attribute In GCC

```
1 int my_function (void) { G C
2     /* "passwd" is a local variable and becomes secret on its definition */
3     char *passwd = get_user_input (); ← Local variable
4     validate (passwd);
5 }
6
7 struct RSA_keys {
8     char public_key[256];
9     /* "private_key" field should be secret for all instances */
10    char private_key[256]; ← Type definition
11 };
12
13 /* "key" will always be secret when the function begins */
14 int encrypt (char *key, char *buf) { ... } ← Function's parameter upon function's entry
15
16 /* "output_key" becomes secret when the function returns */
17 int generate_secret_key (char *output_key);
```

Listing 2.1: Different ways of creating secrets in .

Designing Secret Attribute In GCC

```
1 int my_function (void) { 
2     /* "passwd" is a local variable and becomes secret on its definition */
3     char *passwd = get_user_input (); ← Local variable
4     validate (passwd);
5 }
6
7 struct RSA_keys {
8     char public_key[256];
9     /* "private_key" field should be secret for all instances */
10    char private_key[256]; ← Type definition
11 };
12
13 /* "key" will always be secret when the function begins */
14 int encrypt (char *key, char *buf) { ... } ← Function's parameter upon function's entry
15
16 /* "output_key" becomes secret when the function returns */
17 int generate_secret_key (char *output_key); ← Function's parameter upon function's exit
```

Listing 2.1: Different ways of creating secrets in .

3 **field-sensitive** attributes to identify secrets:

1. secret:

- variables,
- parameters (only applied to **defined functions**):
 - ▶ considered secret upon **function's entry**.

3 **field-sensitive** attributes to identify secrets:

1. secret:

- variables,
- parameters (only applied to **defined functions**):
 - ▶ considered secret upon **function's entry**.

2. secret_out:

- parameters or return value:
 - ▶ for both defined and undefined functions.
 - ▶ considered secret upon **function's exit**.

3 **field-sensitive** attributes to identify secrets:

1. `secret`:

- variables,
- parameters (only applied to **defined functions**):
 - considered secret upon **function's entry**.

2. `secret_out`:

- parameters or return value:
 - for both defined and undefined functions.
 - considered secret upon **function's exit**.

3. `secret_t`:

- type definition (e.g., `typedef` instructions):
 - each instances considered secret at **declaration**.

Identifying Secrets Is Not Enough *Simulating Functions*

- Some functions are known to have **side-effects** regarding secrets:
 - memcpy-like functions: potentially **propagate** them
 - bzero-like functions: **destroy** them
 - *unsafe* functions: **leak** information about them

Identifying Secrets Is Not Enough *Simulating Functions*

- Some functions are known to have **side-effects** regarding secrets:
 - memcpy-like functions: potentially **propagate** them
 - bzero-like functions: **destroy** them
 - *unsafe* functions: **leak** information about them
- No need to have access to their definition:
 - only need to identify them
- Impossible to hardcode them all:
 - **project specific**
 - **context-dependent** (e.g., length of a buffer)

Identifying Secrets Is Not Enough *Simulating Functions*

```
1 void [[propagator("0", "2")]] custom_memcpy  
2     (void *src, size_t src_size, void *dst, size_t dst_size);
```



Listing 4.2: propagator attribute: identify conditional propagators.

Identifying Secrets Is Not Enough *Simulating Functions*

```
1 void [[propagator("0", "2")]] custom_memcpy  
2     (void *src, size_t src_size, void *dst, size_t dst_size);
```



Listing 4.2: propagator attribute: identify conditional propagators.

Identifying Secrets Is Not Enough *Simulating Functions*

```
1 void [[propagator("0", "2")]] custom_memcpy  
2     (void *src, size_t src_size, void *dst, size_t dst_size);
```



Listing 4.2: propagator attribute: identify conditional propagators.

```
1 void [[destructor("0")]] custom_bzero (void *, size_t);
```



Listing 4.3: destructor attribute.

Identifying Secrets Is Not Enough *Simulating Functions*

```
1 void [[propagator("0", "2")]] custom_memcpy  
2     (void *src, size_t src_size, void *dst, size_t dst_size);
```



Listing 4.2: propagator attribute: identify conditional propagators.

```
1 void [[destructor("0")]] custom_bzero (void *, size_t);
```



Listing 4.3: destructor attribute.

```
1 int [[unsafe]] printf (const char *, ...);
```



Listing 4.4: unsafe attribute: identify unsafe functions.


```
1 extern int [[unsafe]] printf (const char *, ...);
2 extern void [[destructor]] erase (char *, char *);
3 int main(void) {
4     char [[secret]] password[256];
5     char * [[secret]] expected = "fubar";
6     int res;
7     get_password (password);
8     if (validate (password, expected))
9         res = 0;
10    else {
11        printf ("Wrong password, expected password was: %s)", expected);
12        res = 1;
13    }
14    erase (password, expected);
15    return res;
16 }
```


Listing 2.5: A vulnerable and naive authentication  code snippet.

```
1 extern int [[unsafe]] printf (const char *, ...);
2 extern void [[destructor]] erase (char *, char *);
3 int main(void) {
4     char [[secret]] password[256];
5     char * [[secret]] expected = "fubar";
6     int res;
7     get_password (password);
8     if (validate (password, expected))
9         res = 0;
10    else {
11        printf ("Wrong password, expected password was: %s)", expected);
12        res = 1;
13    }
14    erase (password, expected);
15    return res;
16 }
```


Listing 2.5: A vulnerable and naive authentication  code snippet.

3. Conclusion

-  and compilers lacks a **notion of secret**:
 - a set of attributes could leverage secret analysis in GCC,
 - without breaking compilation!

-  and compilers lacks a **notion of secret**:
 - a set of attributes could leverage secret analysis in GCC,
 - without breaking compilation!
- GnuSecret's evaluation ongoing:
 - assess false positive and negative rates
 - current approach idea:
 1. generate random C programs
 2. use dataflow analysis to find dependencies
 3. automatize code annotation and run the tool

Takeaway

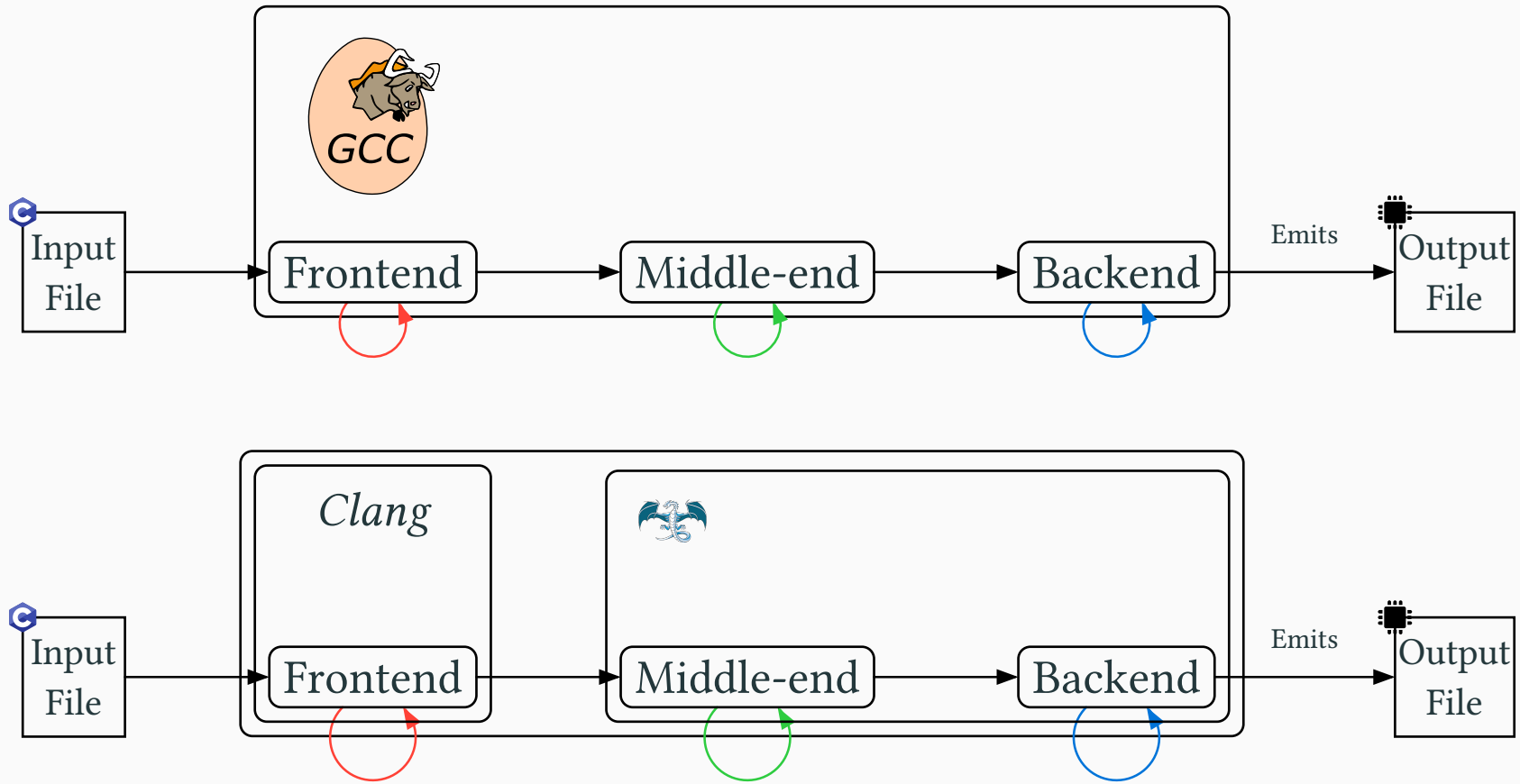
-  and compilers lacks a **notion of secret**:
 - ▶ a set of attributes could leverage secret analysis in GCC,
 - ▶ without breaking compilation!
- GnuSecret's evaluation ongoing:
 - ▶ assess false positive and negative rates
 - ▶ current approach idea:
 1. generate random C programs
 2. use dataflow analysis to find dependencies
 3. automatize code annotation and run the tool

 pierrick.philippe@irisa.fr

 [ricked-twice.github.io](https://github.com/ricked-twice)



GCC vs. LLVM-based compilers



50 Shades of IRs: A GCC Story

