How gcc-Linux-implementation works for code-injection-detection
https://github.com/code-injection-detection/gcc-Linux-Implementation
May-2018

This is an unofficial guide. Many things are temporary and about to change. Also, this is a high-level approach and we do not delve into the details.

Ok let's start:

Introduction

What the code tries to do:

First we need to talk about the general idea of this repo. Its goal is to emulate the secure system described by Lipton, Ostrovsky and Zikas here:
http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=6311

This publication talks about a secure system that defends against an adversary who has arbitrary memory write capability. Specifically, we assume a remote attestator, who periodically queries the system which may be under attack, and verifies that the system has not (or has) been attacked. The power that is given to the adversary is a magic wand (let's say a radiation wand), that allows him to overwrite specific bits of the memory at any time.
Imagine he moves his wand, and the bit at address X is set to 1, and another bit at address Y is set to 0. Or something else. We must point out the fact that these changes are not done through the CPU instructions, but from an outside source (magic wand / radiation / row hammer attacks / etc).
The adversary has some restrictions however: He cannot overwrite something that is in the CPU (only in the memory), and he cannot read the memory before he can write to it. However, if he overwrites a place and he puts his own code there that outputs the rest of the memory, this way he can read it. But he needs to overwrite a place first.

The authors defeat such an adversary by the following algorithm:
They split the useful bytes of the memory into blocks, and among them they interleave keyshare bytes (which together constitute a key) and MAC bytes (more details in the paper). So the memory structure is now

<useful bytes1><keyshares1><MAC bytes1> <useful bytes2><keyshares2><MAC bytes2> … etc …

(the keyshares are actually 2 groups of keyshare bytes, to avoid Time-of-check-time-

of-use attacks in the attestation procedure – more details about that in the paper).

The CPU, every time it fetches a block, verifies the corresponding MAC bytes. If they are wrong aborts the execution. When it writes a block, updates the MAC bytes. Also, periodically, the remote attestator queries the system with a challenge. The system needs to gather all the keyshares for the memory to reply correctly. The attacker, when he overwrites, will either invalidate a specific MAC, or the key that is spread in the memory in the keyshares. The proof of correctness is in the paper, and we won't go into any more details – you should probably read that because the rest won't make much sense.

Here we talk about our implementation, which tries to emulate the performance of such a system.

We simplify a process' memory to have only 4 parts: text (code), data (globals), heap, stack.
We interleave keyshares + MAC bytes among the useful bytes in these 4 parts, and we use special functions (secure getters/setters) to access them.
For the code, we split the assembly into blocks, add NOP instructions (which will be then replaced by keyshare+MAC bytes), add a jmp above the NOPs to reach the next block (we don't want to execute the bytes that are keyshares/MACs), and, depending on the configuration, we add (in various parts inside the block) code that does MAC verification (more on that later).
For the globals, we organize them into a C struct. Among the actual variables, we add keyshares+MAC bytes into that struct too. We use special functions (getters/setters) to interact with the variables.
For the stack, we allocate an array in the normal, unsecured heap. We interleave keyshares + MACs, use getters/setters, and we implement our own calling convention. Instead of calling a function, we push its arguments into our "secure" stack, we push the return address, base pointer and local variables. Then we goto the start of the function. When we return, we goto the return address (and pop the stack frame).
For the heap, we allocate another array in the heap. Same story, keyshares+ MACs, getters/setters. We implement (using the secure globals and the secure stack) our own version of malloc/free, which keep two doubly linked lists, one for the allocated chunks and one for the free. Basically we copy the idea of dlmalloc (http://g.oswego.edu/dl/html/malloc.html) , and use it in our secure heap, without optimizations.

Great. We have described what the code generally does.

At this moment however, there are two things that we should note:
One is that, since the MAC calculation is very expensive to be done every time, we introduce the idea of a secure cache inside the CPU (which cannot be overwritten by the adversary since it is in the CPU). Using that cache, when the CPU fetches a block,

after the verification it is put into the cache. If that needs to be fetched again, the CPU ignores what is in the memory and only reads what is in the cache, without verification. Same idea with writing to a block, it will be brought into the cache first. The blocks may of course leave the cache (when replaced by other blocks), and when they do they go to the normal memory and overwrite the old block that was there. The cache helps a lot in the benchmarks, since it allows us to skip many MAC verifications.
The second thing to note is our benchmark strategy. We emulate the secure system in software. That means that our secure getters/setters, every address calculation (to take into account that there are keyshares/MAC bytes in the memory), the jumps above the keyshares/MAC bytes in the code etc, are done in software. That incurs a slowdown, which we call baseline slowdown. Even if we optimize these code parts, they will still be slower than how the specialized hardware would implement them. We know this and we note this – our baseline slowdown is a pessimistic estimate on how slow this whole thing would run.
There is also another reason for slowdown – the MAC calculation. We have to save the program state, calculate the MAC, and restore the program state. The MAC calculation itself is done using hardware (more on that later), but the save/ restore state procedure is not (and some parts of the MAC verification) – and that is destroying our performance, especially if we have many MAC verifications. Of course, the secure hardware would have a special circuit that calculates the MAC, and no state change would be needed. So, what we do, is calculate the *number* of MAC verifications a certain program will do, and (offline) we calculate the time it takes for one MAC verification. Using that, we can extract the time it takes for all the MAC verifications – and if we add the baseline slowdown, get a better (still pessimistic) estimate of the slowdown of the whole thing.

Now we will move on to talk about how each part is secured. We won't delve into our code just now, we will first describe the ideas of implementing.

Before we move on however, it would be a good idea if you tried to run our code.
Make sure you have the dependencies installed : A java compiler, gcc, libssl-dev , python3-pycparser , binutils.
Now go to **code/** and run **./automate.sh** . It tells you it needs some parameters, and gives an example on how to execute it. Do it, and after a minute or so (for the subsequent runs it takes significantly less), it will produce an executable. You can run it (**./main_program_ksec**) and observe the output. Apart from the executable, you will see it has created a lot of other files – thy are mainly the copied templates because we can't touch them, but we can alter their copies.

How the various parts of the memory are secured

How the code is secured:

Now we are going to talk about how the code is changed, in more detail.
Let's assume that we have a given program that already initializes the secure heap and stack, and uses our secure convention of calling the functions, and the secure getters and setters in C language. That program must be compiled in a way that its code will be secure, too. We have not talked about the securing of such structures, but for now assume that that's how it is.

There are some parts that do not have to be secured, however. The functions that implement the mac initialization, verification or the secure getters/setters would normally be implemented in hardware. So we have them in separate files which are compiled with all optimizations enabled, and afterwards linked with the rest of the program.

In the end, we have a .c file (main_program.c in **code/** directory) which includes other .c files and .h files, and is linked with fully optimized object files. That compilation unit has all the secure code, and is being assembled into main_program.s .
So now we have all the secure code inside an assembly file, and it's time to insert the verification code, the keys and the macs. That is the job of **src/ Secure_Assembly_v2.java** (outside of **code/** directory) – which implements the following idea:

Consider some assembly code:

```
        ...
        movq    -24(%rbp), %rax
        cmpq    -40(%rbp), %rax
        jl      .L678
.L665:
        movq    -32(%rbp), %rax
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rax
        addq    $88, %rsp
        ...
```

That code, as the scheme dictates, must have keyshares+MAC bytes interleaved among the actual bytes of the code. If we simply add the bytes, the execution environment will simply try to run these "random" bytes and will crash. That will not happen however, if we insert jumps above the extra bytes. Note: At first, we insert one-byte NOP commands for the keyshares+MACs, they will be replaced later however by th actual bytes.
There is also another problem. X86 instructions are variable size (typically 1-15 bytes

long), but our scheme requires fixed size blocks, We solve this by padding extra NOPs after the jmp.
So, the code would be like that (the block splitting is arbitrary):

```
        …
        <previous_NOPs>

.BLCK_1500:
        movq    -24(%rbp), %rax
        cmpq    -40(%rbp), %rax
        jl      .L678
.L665:
        movq    -32(%rbp), %rax
        jmp .BLCK_1501

        <NOPs for padding>
        <NOPs for keyshares>
        <NOPs for MACs>

.BLCK_1501:
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rax
        addq    $88, %rsp
        jmp .BLCK_1502:

        <NOPs>
        ...
```

There is one minor problem. Some commands (the jump command family) are not fixed size – their size depends on the offset (which changes depending on how many blocks away their target may be). We can enforce a 32-bit offset by adding a .d32 at the end of the opcode: jl .L678 → jl.d32 .L678
We also add, after each jump that goes to the next block, some code canary bytes, and some metadata that help us when verifying the MAC. Specifically, we add:
a) 3 canary bytes with value 0x42, to tell us “the jmp before me is a jmp above keyshares+MACs”
b) 2 bytes that denote the size of the code bytes, from the .BLCK_x label to (including) the jmp .BLCK_(x+1).
c) 2 bytes that denote the number of padded NOPs.

So the code becomes:

```
        …
        <previous_NOPs>

.BLCK_1500:
        movq    -24(%rbp), %rax
        cmpq    -40(%rbp), %rax
        jl.d32  .L678
.L665:
        movq    -32(%rbp), %rax
```

```
        jmp.d32 .BLCK_1501
        <canary bytes>
        <metadata bytes>
        <NOPs for padding>
        <NOPs for keyshares>
        <NOPs for MACs>

.BLCK_1501:
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rax
        addq    $88, %rsp
        jmp.d32 .BLCK_1502:
        <canary bytes>
        <metadata bytes>
        <NOPs>
        ...
```

There is one major problem that remains however. While the hardware is supposed to know where each block resides and run the MAC verification every time it is accessed, we only have our emulation code that is executed – no extra MAC verifications. So, we must also add verification code whenever we believe that the code block is accessed.
Our verification code should do the following:
a) Save the current execution state (registers)
b) Check if we need MAC verification (we may not need if the block is in the code cache inside the CPU, we only need if it is not and it is added to the cache)
c) If yes, verify the MAC. If not, do nothing
d) Restore the current state

We have a function that does the checks and verification, do_verify_code_on_the_fly(). It emulates what the secure hardware would do. Its call consists of 3 assembly commands:

```
pushfq
call do_verify_code_on_the_fly
popfq
```

(the push/pop f commands are there because the first command of the function is a sub, which clobbers the flags register)
That code can be put practically anywhere, since it does not necessarily verify the MAC, but checks if it needs to be done. So, we put it after every suspicious point that may trigger a MAC verification:
a) After function starts
b) After (normal) call commands (except for the functions that emulate functionality done in hardware). Our secure stack does not use the “call” command, it just goto’s. But we include this case for completeness.
c) After labels
d) At block starts
e) We also place the code BEFORE a jump to register, and we check if that jump will

land us in another block, and if yes, we verify it too.

That’s how the code becomes:

```
        …
        <previous_NOPs>

.BLCK_1500:
        pushfq #start of block
        call do_verify_code_on_the_fly
        popfq
        movq    -24(%rbp), %rax
        cmpq    -40(%rbp), %rax
        jl.d32    .L678
.L665:
        pushfq #after label
        call do_verify_code_on_the_fly
        popfq
        movq    -32(%rbp), %rax
        jmp.d32 .BLCK_1501
        <canary bytes>
        <metadata bytes>
        <NOPs for padding>
        <NOPs for keyshares>
        <NOPs for MACs>

.BLCK_1501:
        pushfq #start of block
        call do_verify_code_on_the_fly
        popfq
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rax
        addq     $88, %rsp
        jmp.d32 .BLCK_1502:
        <canary bytes>
        <metadata bytes>
        <NOPs>
        ...
```

All this added code incurs a heavy performance penalty. For everything except the verification, that’s more or less the best we can do, because we emulate the system in software. Of course, specialized hardware wouldn’t even have the jumps, it would know where each block ended and it would jump over the keyshares and MACs automatically.
For the verification code however, we do what we described in the Introduction: We can only count the number of the MAC verifications. After the execution, we can calculate the time it takes for one MAC verification, and this way find how much time all the MACs took. We can then add the time it takes for the code without including the MAC verification code, and extract the final time this way.

So now we have an assembly file (main_program_sec.s) which has blocks of useful code, on-the-fly verification, jmps and padded nops. The next step is to replace the

NOPs for keyshares+MACs with the proper bytes.
We assemble that assembly file into main_program_sec (an executable), and we iterate over it to replace the bytes.
That is the job of **src/Secure_Machine_Code_v2.java** , which runs through the main_program_sec, finds the jmps which have X nops after them, and replaces the keyshare bytes and the MAC bytes.
It also generates the keys for the secure stack and heap, and writes them to a file so that they can be initialized by the program when it runs. After that procedure the program (main_program_ksec) is ready to run.

How the globals are secured:

The secure global variables all reside in a struct. That struct is declared in **template_files/headers_needed_template.h .** After each useful variable, we pad with useless bytes to reach a full block (we don't group many global variables together in one block), and we also add keyshare and mac bytes. After the compilation (run of code/automate.sh), we can see that the code/main_program.c file has the initialization of a big global struct.
We access the globals using special getters/setters which take into account the fact that the MAC has to be verified/updated.
The way we do this right now, every block of <useful_bytes>||<keys>||<macs> can only hold only one global variable. Should someone want to use arrays of variables, he has to use the heap. In the heap, he can allocate the array he wants.
Here's the start of the global struct:

```
typedef struct global_variables_struct
{
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:double
double global_double_variable_for_testing;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int test_global;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:int
int secured_i;
//ATTENTION: GLOBAL VARIABLE FOLLOWING! | SIZE:long
long secured_sum;
….
```

You can see the annotations with the sizes. These help a python script that runs afterwards to initialize the proper number of bytes.

How the stack is secured:

The secure stack is an array allocated in the normal, "insecure" heap. We use two global variables which function as registers (base pointer and stack pointer), and, starting the stack pointer from the end of the stack, we decrease it whenever we need to allocate something into the secure stack (and increase when we deallocate).
Like the globals, we interleave keyshares+MAC bytes among the useful bytes, and

we use specific getters/setters for accessing the useful bytes (that pay attention to fetch/set the correct bytes and not the keyshare ones/MAC ones , as well as to verify/update the MAC when needed).
In this secure stack we implement our own calling convention, similar to the ones that are used in real applications. When we encounter a “function call”, we call the stack allocator to allocate a function frame (with the size we want), we call the setters to set the function arguments, return address, base pointer, possibly stack canary, and local variables that need initialization. We also reserve some space for the return value, if the function returns something. The jump to the function code (and out of it) is done with a goto. In order for this to work in C, the code of all our tests is in one big function, great_function_that_wraps_the_tests() , in **template_files/ tests_with_new_stack_template.c .** We need to do that because we cannot jump to a label that is outside our function.

Here’s some code that uses the secure getters/setters of some variables that reside in our stack (note that their names have been previously defined as an offset from the base pointer):

```
   for ((int)SET_STACK_INT( i , 1); (int)GET_STACK_INT( i ) <= 10; (int)SET_STACK_INT( i , (int)GET_STACK_INT( i )+1))
  {
          (double)set_stack_double_array_element( min_heap , (int)GET_STACK_INT( i ) , 10 - (int)GET_STACK_INT( i ));
       (int)set_stack_int_array_element( min_heap_corresponding_indexes , (int)GET_STACK_INT( i ) , (int)GET_STACK_INT( i ));
                (int)set_stack_int_array_element( fixed_indexes , (int)GET_STACK_INT( i ) , (int)GET_STACK_INT( i ));
  }
```

Also, here’s some code that calls a function:

```
({
//allocate space
void *returned_addr_after_allocating_1002=allocate_mem_into_secure_stack_in_chunks(8);
if (returned_addr_after_allocating_1002==NULL)
        {printf("ERROR! no stack mem left -> line %d\n",__LINE__);exit(8);}
//set return address
set_stack_pointer_array_element(returned_addr_after_allocating_1002-(3+1+1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data),1,&&return_label_mh_combine_no_3);
//create a normal array that will be inserted into the secure stack, and will hold the arguments
long size_of_array_for_array_fun_parameters_1003=2*4;
int array_for_int_fun_mh_combine_params_1003[2];
array_for_int_fun_mh_combine_params_1003[0]=(int)GET_STACK_INT( new_ind );
array_for_int_fun_mh_combine_params_1003[1]= (int)GET_STACK_INT( num_of_elements );
//put that array in the secure stack
insert_data_into_stack_mem(size_of_array_for_array_fun_parameters_1003,(unsigned char*) array_for_int_fun_mh_combine_params_1003, (unsigned char*)returned_addr_after_allocating_1002-(3-0)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data));
//do the same for the pointer parameters
long size_of_array_for_array_fun_parameters_1004=3*8;
void * array_for_pointer_fun_mh_combine_params_1004[3];
```

```
array_for_pointer_fun_mh_combine_params_1004[0]=(double*)GET_STACK_PTR( min_heap );
array_for_pointer_fun_mh_combine_params_1004[1]= (int*)GET_STACK_PTR( corresponding_indexes );
array_for_pointer_fun_mh_combine_params_1004[2]= (int*)GET_STACK_PTR( fixed_indexes );
insert_data_into_stack_mem(size_of_array_for_array_fun_parameters_1004,(unsigned
char*)array_for_pointer_fun_mh_combine_params_1004,(unsigned char*)
returned_addr_after_allocating_1002-  (3-1)* (stack_bytes_used_for_keyshares +number_of_mac_bytes
+stack_bytes_for_useful_data));
//set the base pointer in the stack to hold the former value of the “register”
set_stack_pointer_array_element(returned_addr_after_allocating_1002- (3+1+1)*
(stack_bytes_used_for_keyshares+ number_of_mac_bytes+ stack_bytes_for_useful_data), 0,
base_pointer_for_stack);
//set a new value for the “register”
base_pointer_for_stack=returned_addr_after_allocating_1002-(3+1+1) *(stack_bytes_used_for_keyshares+
number_of_mac_bytes+ stack_bytes_for_useful_data);
//goto the function code (the locals are set in there)
goto mh_combine_start_label; return_label_mh_combine_no_3:  ;
//get the return value and use it as the return value of this big block of code
get_stack_int((last_unused_stack_memory-
(3+1)*(stack_bytes_used_for_keyshares+number_of_mac_bytes+stack_bytes_for_useful_data)));
});
```

All this code is generated by python scripts. The way they know what to allocate each time, the number of arguments, local variables etc (everything that such a script would need to know), is done in two ways: The first (old one), is by annotating all that info before the function start. The second (current one), is by writing normal C code and parsing it with the pycparser tool, which allows us to extract all this information. Some examples can be seen in **code/template_files/ tests_with_new_stack_template.c**

How the heap is secured:

Like the stack, the “secure” heap is an allocated array in the normal heap. We interleave keyshares and MAC bytes, and we use special getters/setters for access. Using the fact that we now have secure globals and a secure stack, we have implemented our own versions of secure malloc and secure free. We have taken the idea from Doug Lea (http://g.oswego.edu/dl/html/malloc.html) and , with minor modifications, we have implemented it for our secure heap. We did not write any optimizations since it would take ages, and it would be a completely different direction that the one we wanted to follow (emulate the secure system). If someone wants to benchmark many normal malloc invocations versus our version, it is fair to use the insecure implementation that we have written for our own version, and not the libC malloc() . Of course, for a small number of malloc’s (that do not alter the overall time) one can use whatever he wants. The secure implementation of malloc is in **template_files/ heap_manager_functions_that_use_secure_stack_template.c** .

Two words about the MACs

A typical block structure is the following:
<useful bytes> || <keyshare 1> || <keyshare 2> || <MAC bytes>

Normally the MAC algorithm has to defend against an adversary who has overwritten less than polylog(<size_of_keyshares>) bits from the keyshares (and everything else he wants from the block). Such an adversary should not be able to forge a valid MAC. An algorithm that satisfies that property (as described in the paper) is
MAC=w * H(K1) + H(K2)
where:
w=useful bytes
H() = an ideal hash function
and the multiplication and addition are done modulo 2^(<size_of_MAC>).

We know that the secure hardware, whether it implements that scheme or something else that is deemed secure, will have it implemented in hardware (in order to increase efficiency). The only hardware that we have however, are the AES-NI extensions in our Intel CPUs. So what we do, is that we use AES in CBC mode as our MAC scheme.
We need to make some important clarifications for this scheme however: The randomness of the MAC is provided by the keyshares in the block. The key of the AES algorithm is a fixed (known) value, and the randomness of the output is because of the keyshares that are being encrypted. The MAC is the final block of the AES-CBC encryption of <useful bytes> || <keyshares>.
We have to assume that AES has the properties we want (since that’s the only hardware that we have). Again, we want the MAC that such a scheme outputs to have the following property:
An adversary who knows where a valid block is, and does not know the keyshares and MAC bytes of that block, if he overwrites some bits of the useful bytes, some bits of the old MAC bytes, and less that polylog(<size_of_keyshares>) bits of the old keyshares, the resulting MAC should be invalid. If he overwrites more than polylog(), the MAC may be valid but he will have lost the keyshares and the remote attestation will fail.

The code in more detail

Here's a simple overview of the directory structure:

Directory structure:
1) src/: Inside there are the java source files than manipulate the assembly and the final executable.
2) code/: In there are the other files that emulate the secure machine. There's a lot of things there. In the **template_files/** we have the templates of the code that are copied and changed accordingly (as the configuration of the compilation demands). In **pycparser_scripts/** we have some python code that transforms C code → C code that uses the secure structures (does not support the full C spec but it's usually enough). In **scripts_for_code_generation/** we have some other scripts that help for the emulation of the secure system. The other directories are not that important.
3) bin/: This directory does not exist on github. In there there are the java class files (that Eclipse produces automatically). Someone that does not use Eclipse should compile the .java files in the src/ directory and move them inside the bin/ directory. To save the hassle, a script does it automatically, if the java compiler (javac) is installed.

Some constants of our implementation:
Our implementation has many parameters that can be configured. Other parameters are fixed and cannot be altered. The parameters can be seen in **template_files/ headers_needed_template.h**. The parameters that can be altered are done so by code/ automate.sh .

Here we will only talk about the most significant parameters (some of them not subject to change).
- Keyshare size: Each keyshare is 16 bytes long (128 bits). In one block, we have 2 keyshares, so 32 bytes in total.
- MAC size: The MAC size is the AES block length, 16 bytes (128 bits).
- 3 canary bytes are used after the jump to the next block.
- 2 bytes are used after these canaries to denote the length of the useful bytes (+ verifications + jmp).
- 2 bytes are used after them to denote the length of the padded NOPs.
- 7 bytes are used for every verification call (push, call, pop)
- The added bytes per code block (jmp to next block, verifications, metadata after the jump) are not included in the MAC calculation (the specialized hardware wouldn't even have them there). The padded NOPs are included.
- Possible sizes of useful bytes per block are only multiples of 16, because the AES block size is 16 and padding is unnecessary overhead. The size of the useful bytes is the same in code, globals, stack, heap. The code may have extra bytes added by us, but these extra bytes are ignored.

- The size of the stack canaries (if they are used), is 8 bytes.

Description of the most significant files:

First, we have a .c file that contains ALL of the code (except the functions of the standard C library that it #includes). That code is spread among different C files which are included by that main file. That code includes the main_program_function() function (that's practically the main, except some crypto initializations, so the actual main() has to be somewhere else), test functions, the secure heap implementation, the secure stack implementation and the verification procedure.
That program is compiled in its final "secure" form after some changes are done to it. So we actually work with the template of that program, to which we make changes through scripts and then compile.

Let's now talk about the significant files.

Let's open **code/template_files/** directory and take a look at **initializer_template.c** . There we can see some initializations in our main() function. The actual job however is done in the main_program_function(), which is the one that is secured. Main() function is not secured, since someone has to initialize the MAC calculation functions, and the other tools that we use.

Now let's look at **main_program_template.c.** It references the secure globals declaration in headers_needed_template.h (take a look at the python calls), we can see that it includes a file called "functions_needed_header.c". After that there is the main_program_function() and then another file is included, the "functions_needed_footer.c". These two files include every other function that may be needed. If we want to peek inside, we have to open **functions_needed_header_template.c** and **functions_needed_footer_template.c** .
That is not necessary. What needs to be understood are two things: a) Inside our template files there are calls to python scripts that change the code, so the final program that is compiled is the output of the python scripts (which pad the templates with proper C code), b) Everything that is to be secured is in one compilation unit (since we use all those includes), c) The template files are copied (often with some changes) to the non-template equivalents, and the compilation is done in the latter. The code that a programmer will write should be in the templates though.

Now let's delve deeper into the securing procedure, explaining it step by step. The core of that procedure is a bash script, **code/automate.sh** . What it does in summary: Takes the user parameters, changes the templates using python, compiles the program into assembly, pads NOPS among the assembly commands, assembles into executable, and changes the NOPS with keys and macs inside the executable.

Let's open it and see it. In the start we can see the parameter fetching (and setting), some error checking and compilation of the java files if they are out of date. Then we

see a line which has the string “python3 **set_correct_defines.py**”. Let's go ahead and open that python file, inside **scripts_for_code_generation/** directory. It consists of two parts. The first one sets the proper parameters in the file “headers_needed.h”, and the second one is obsolete (is used for the former stack implementation, so we don't have to pay special attention to it).
The file **template_files/headers_needed_template.h** is a header file included by almost every file in the project. It has many compile time parameters (like the size of the heap and the stack, the number of the keys etc), some typedefs and the names/types of the global variables. As we can see, the python script alters some of these parameters, as the user requests. It writes to the file “headers_needed.h” inside the **code/** directory. After that, every unit includes the non-template file “headers_needed.h” which has the proper parameters.

Let's move on inside the bash script now. We see that some template files are copied into their non-template counterparts. These ones contain the heap manager functions, and the tests that we run. Every test eventually gets added into the **tests_with_new_stack.c** file, using the “sed” command. That file, however, has not taken its final form yet. In order to see what has been put to be tested, open **template_files/tests_with_new_stack_template.c** . You can see that there is C code, but there are some parts that ask python to call a function. That code part is later replaced by python to contain valid C code.
You will also see some commands in automate.sh about the C reconstructor. That is an experimental procedure that tries to take normal C code, and transform it into code that uses the secure structures that we have (globals, stack, heap), communicating with them using secure getters and setters. We use the pycparser package for the job, and we support a limited subset of C (which is enough to write and reconstruct some programs, but not all of them). We won’t delve into the internals of the reconstructor, since it is a PL topic. All we need to say is that the code of the reconstructor is in **code/ pycparser_scripts/ .**

However, this procedure allows us to see how the secure code looks like, and the phases of the reconstruction. Please open **tests_for_stack_commands_supporting_ast_parsing.c** . There, you can see normal C code that is also used for the insecure tests. After the pass of the C reconstructor , the output is in **1st_pass_of_C_reconstruction_of_tests.c .** There, you can see the declaration of the globals with annotations, and all the rest that uses secure getters/ setters. When a function is being called, we use an annotation of the type
{{{ HEY PYTHON CALL FUNCTION WITH NEWER TEMPLATE: <name_of_fun> | HELPING ARGS FOR FUN CALL: <possible_metadata> | PARAMETERS TO CALL WITH: <params> }}}
That also needs to be changed to hold code that manipulates the secure stack. That is the job of the script **scripts_for_code_generation/ insert_stack_manipulation_commands_given_ast_of_functions.py** . After it finishes, the result is valid C code that uses the secure structures that we have. Open **2nd_pass_of_C_reconstruction_of_tests.c** to see the output of that script. You will

probably notice the many #defines – they are actually the names of the variables as offsets from the base pointer. Finally, we have the script **scripts_for_code_generation/ transfer_c_reconstructor_globals.py** that isolates the global declarations that have been created, and transfers them to headers_needed.h .

Let's continue in automate.sh:
We see that the file **insert_new_stack_commands_groups_of_vars_as_arrays.py** is called. Its job is similar to the script that created the stack commands above. The only difference is that it uses the old approach, with annotation that describes the function metadata (the new approach uses the output by pycparser on normal C code). You can see the annotations in the file **template_files/tests_with_new_stack_template.c .**
A note: since we had many revisions of the way a function may be called, you will also see annotations of the old version:
//HEY PYTHON CALLING FUNCTION : <name_of_fun> |PARAMETERS TO CALL WITH: <params>
While this works, it does not allow for function calls in a function's parameters.
Now the file **tests_with_new_stack.c** is in its final, "secure", form and you can open it and see all the changes that the above scripts have made. It is now ready to be included by the main program.

Moving on inside automate.sh, we see that some more template files are copied inside **code/** directory.

The next step is to parse the annotations of the globals, an create the struct that will hold them (which will also have the keyshares and the MAC bytes).
That is the job of the script **scripts_for_code_generation/ insert_keys_and_macs_among_globals.py .** The first time it is called only reads the headers_needed.h file, and creates the global struct there. That is to make a functional header file, which will be used in the subsequent compilations. We must note the fact that this script normally reads other files too for global declarations (there shouldn't be any, we leave them there for compatibility reasons), but most importantly, also creates code that takes into account the global keyshares when the keys are gathered (that code is created in other files). For now however, it only creates the global struct in the headers_needed.h file.

Now that we have a functional header file, we can compile the files that will emulate the hardware circuits. Of course, we enable optimizations for them. For the on-the-fly verification function, we compile it and then, depending on the location of its local variables, we adjust the offset that fetches the return address from the stack.
The **template_files/ crypto_functions_template.c** holds this function, along with the MAC calculation functions, and the cache calculation ones.
These MAC calculation functions should be moved into the file **template_files/ mac_verification_functions_template.c ,** it is in the todo list.
The files **calc_mac_for_external_programs*.c** are files that calculate the MAC but

are used by the code generation procedure to replace the NOPs with MAC bytes.
The file **mac_time_calculator.c** is used to calculate the time it takes for the MAC calculations (offline).
The **sha256*** files hold a software implementation of the sha256 hash function, we don't use them. They are there for legacy reasons.
We create their object files, in order for them to be able to be linked along with the secure code.

The next step is to reconstruct the headers_needed.h file again, but this time create the code that can check the global MACs at the end (for correctness), and take into account the keyshares of the globals when we gather the keys. As we said, that is the job of **scripts_for_code_generation/ insert_keys_and_macs_among_globals.py** . Another thing it does is copy some other template files that are specified in it to their non-template counterparts. We leave them to be processed by it (and not just "cp" the templates) for the case that they might contain global variable declarations (they shouldn't, but there is old code we don't want to break). As you can see, the two last files that it processes (**template_files/ functions_needed_footer_template.c** and **template_files/ verification_procedure_template.c**) , contain the final MAC check and the keyshare gathering.
We can also see some other output of the script if we open **headers_needed.h** (we see the global struct with the globals, keyshares and MACs), and **main_program.c** (we see the global initialization numbers).

We then compile the secure getters and setters with optimizations enabled (they would be circuits after all), and we compile (through the **Makefile**) , our program. The files that have been compiled with optimizations do not have their code secured (i.e. keyshares and MACs interleaved). The one that is secured is the main_program.c , which #includes the other C files it needs. The output of the compilation is the **main_program.s .**

The next step is to secure the assembly code. That is the job of the program **src/ Secure_Assembly_v2.java** and it does what we described in the section where we discuss the code securing process. The output of that execution is the **main_program_sec.s** . If you open it, it becomes pretty clear how the code is secured.

This assembly file is then compiled, and all that remains is to replace its NOPs (except the padded NOPs to reach fixed size blocks) with keyshares and MACs. That is done by **src/ Secure_Machine_Code_v2.java** , which iterates through the executable and replaces the bytes accordingly. The result is the executable **main_program_ksec**, which can be executed and benchmarked.