# Software Integrity Protection Lab
## Documentation Phase 2

Alexander Küchler and Martin Morel

Technical University of Munich
Department of Computer Science
kuechler@in.tum.de, martin.morel@tum.de

## 1 Docker image

The public docker hub site can be found at this link: **Create new docker hub site** https://hub.docker.com/r/kuechlera/labinstrumenter/

You can pull the image with

```
docker pull kuechlera/labinstrumenter
```

## 2 Individual effort

Alexander:

– 

Martin:

–

## 3 Tool usage documentation

**This might change...**
Our solution consists of the following components:

- LLVM Function Pass (`FunctionPass.cpp` ), building the call graph, injecting code to build a shadow stack and to verify it whenever a sensitive function is entered.
- Stack Analysis functions (`StackAnalysis.c` ) that perform the verification of the stack trace, linked together with the object file built from the input source file after having been processed by the Function Pass.
- Shell script (`compile.sh` ) compiling and calling the former components with the correct parameters.

**LLVM Function Pass.**
The Function Pass is loaded using `opt-3.9`, it has a required option `-i`, used to specify the path to the file in which to find the list of sensitive functions. One function name per line are expected in this file.
For example, to modify the bitcode of an input C file named `src.c` and verify the functions specified in file `list.txt`:

```
clang-3.9 -emit-llvm src.c -o src.bc
opt-3.9 -load code/libFunctionPass.so -i list.txt -functionpass < src.bc > src_pass.bc
```

**Stack Analysis.**
In the `StackAnalysis.c` file are defined functions to build the shadow stack and verify it against known traces at runtime. It expects to find a `graph.txt` file containing the list of edges of the call graph generated by our Function Pass. An object file is assembled from this source file and linked with the source code to be protected by the `compile.sh` script presented below.

**Shell Script.**
The `compile.sh` script is provided to ease the use of our solution. It calls the components presented above and produces a protected binary. It expects the following arguments: path to the source file to protect, path to the desired protected binary, list of sensitive functions. To protect the functions defined in `list.txt` of the `src.c` source file and create a new binary named `bin` in the current directory:

```
./compile.sh src.c ./bin list.txt
```

**Integration in the Docker Image.**
The sourcecode, makefiles, binaries and scripts can be found in the `/instrumenter` directory of the docker image. The script `compile.sh` is located in the directory `/instrumenter/build` together with a compiled version of the llvm pass. The sourcecode can be found in the directory `/instrumenter/code`.

## 4 Design decisions

In this section, we describe the design of our LLVM pass as well as the stack verification.

### 4.1 Injected Code

To grasp the stack we want to verify, we rely on building a shadow stack and thus are independent from the system's stack trace and any system calls which might be prohibited or hooked by the user. The code can be found in `StackAnalysis.c` and consists of four major additional functions:

- `void registerFunction(char*)` pushes a new function name to the shadow stack
- `void deregisterFunction(char*)` pops the latest function from the shadow stack if the function name matches the argument
- `void verifyStack()` checks if the stack is valid. If any unexpected function call is detected, it performs a response mechanism
- `void response()` implements the response mechanism we perform in case an invalid stack trace is detected

### 4.2 LLVM Pass

Our solution relies on a Function Pass. This path iterates over all functions that can be found in the input file to protect. We use the path to inject a function call at the beginning and end of every function defined in the input source file. These functions push and pop the function's name to a shadow stack. If the function which is instrumented can be found in the list of sensitive functions, we also inject a call to the function responsible for the verification of the shadow stack previously built at the entry point of each sensitive function. Finally, we inspect every function instructions and look for calls to other function to build a call graph.

### 4.3 Stack verification

To verify if the stack trace is legitimate, a couple of approaches are possible. In this subsection, we discuss some ideas and explain our implementation.

For the stack verification, two criteria are important. First, it should be precise and thus neither introduce false positives nor false negatives. Second, it should be efficient with regard to memory and time consumption because it is performed on runtime of the program.

A first idea makes use of hashing the stack trace and compare it against known good values. This works fine for programs without loops in the call graph. However, it cannot be applied on programs with loops in the call graph because the known good hashes would explode and require an infinite number of hashes. It is possible to reduce the number of hashes by additional analysis on the call

graph and only using the first occurrence of the function in the stack trace. Still, it requires a large number of hash values for a big call graph with many paths and different loops that lead to a sensitive function.

While this allows an efficient verification, it is easy to attack this scheme. It requires complex graph analysis which is in general imprecise, so that a couple of false negatives (that is the verification might fail for valid stack traces). Also, it is easy to find false positives (that is a valid verification for invalid stack traces). E.g. if the functions $f_1, f_2, f_3$ are a cycle in the call graph and function $f_2$ has the right to access a sensitive function. In any case, the stack contains $f_1, f_2, f_3$ which has to lead to a valid verification. However, this is also possible if the loop has been passed and $f_1$ tries to access the sensitive function. With the technique explained above, it is impossible to detect this invalid stack trace.

Note that this holds for any kind where hashing is applied and thus also affects an implementation based on Merkle trees.

As a call graph can be arbitrary complex, another possibility is to check the stack against a known good call graph. In contrast to hashing, we can ensure to be precise and neither introduce any false negatives nor false positives. Therefore, this approach as a better precision than hashing. However, the efficiency of the verification depends on the size of the stack trace and the functions which are stored in the call graph. Hence, we only store the edges which are on a path that can actually reach a sensitive function. Also, it requires less computations while building the stack trace because we do not have to remove any edges.

In summary, there is a tradeoff between performance and precision. As we expect only a small difference in the overhead for the second approach but can ensure a way better precision, we decided to use the second approach.

We use an adjacency matrix to model the known good call graph. As soon as we enter a sensitive function, we check that every transition in our shadow stack is known.

### 4.4 Response model

In this subsection, we discuss some potential response models and explain the choice we made.

Our work is motivated by the search for a method to protect sensitive functions from control flow hijacking. Exiting is therefore a natural, simple and generic response model to choose. As required, it prevents the access to sensible functions when the control flow is abnormal. But here again, it might disclose information about the actual exit point of the program and disclose valuable parts of the defense mechanism to an attacker.

An additional layer of obfuscation would therefore probably be necessary to increase the difficulty for an attacker to reverse our defense scheme. One might also for instance use a set of decoy functions to which we could redirect the control flow in case of an attack. We could also add some reporting functionalities to our response scheme e.g. by sending the malicious stack trace observed to a trusted server.

Such improvements however appear specific to the application to protect. We therefore ship our solution with a the possibility to extend the response to any desired technique. Therefore, it is only necessary to implement the function `response()` in the file `StackAnalysis.c`.

## 5 Performance Evaluation

### 5.1 Memory Consumption

In contrast to the original program, we manage the shadow stack on runtime of the program and also verify the stack trace against a known good execution. We therefore introduce memory overhead to the application. In order to keep the solution usable, this overhead needs to be as small as possible. To test the memory consumption of our implementation, we wrote a simple and small test program. Effectively, it only contains of a main-method and 3 methods that recursively call each other until a counter expires, then a sensitive function is called. This program presents the worst case for our tool as it hardly consumes any memory, every path leads to the call of a sensitive function and the shadow stack always increases.

In an experimental setup, we traced the consumption of the heap using valgrind and the memcheck tool. The results are shown in Figure 1. The red line shows the memory used by the instrumented program whereas the blue line shows the original one. We ran the program for increasing number of loops and could detect that the additional memory consumption for increasing loop count remains small. However, even at the beginning, the maximal memory consumption is around 10 times as high as for the original program. Manual inspection of the traces revealed that the allocation explodes when we read the edges from of the file and store them in the adjacency matrix.

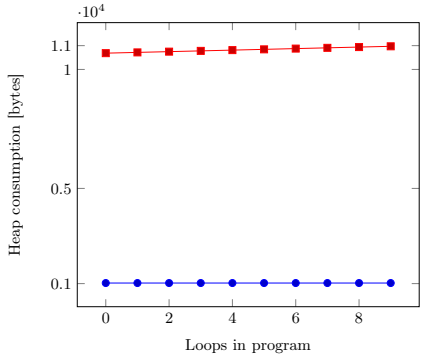


**Fig. 1.** Used heap for test program

## 6 Security Evaluation

The defense mechanism that we implemented essentially relies on a shadow stack built during runtime and further checked against a known call graph, which introduces several vulnerabilities and can be tricked by an attacker.
As the functions used to build the shadow stack are respectively inserted at the beginning and end of every function, they can be easily found by an attacker and then disclose the shadow stack which can be further modified.
On the other hand, to ensure that the call graph, written in a plain text file is not corrupted, we rely on a SHA256 digest. If an attacker gets access to this digest in the binary, she can completely defeat the whole scheme by modifying the reference call graph as she pleases. An additional layer of obfuscation or the use of white-box cryptography seems therefore necessary to make to reversing task more complicated.

An attacker might also defeat our scheme by bypassing the calls to the shadow stack building functions. In the `readKey` example, an attacker can jump from any function to the `readKey` function provided that the destination is located after the call to the `registerFunction`.

The protected binary generated is vulnerable to various static and dynamic patching attacks. The use of a self-checksumming protection tool can therefore provide additional security and make an attacker's task even more difficult.

A summary of the attacks and possible corresponding countermeasures is shown in **??**.