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: `https://hub.docker.com/r/kuechlera/stacktracer/`

You can pull the image with

```
docker pull kuechlera/stacktracer
```

2 Individual effort

Alexander:

- First implementation of FunctionPass
- Implementation of graph
- Implementation of shadow stack
- Conceptualization of stack analysis
- Performance optimizations of stack analysis
- Memory overhead analysis
- Automated attack

Martin:

- First attempt at implementing LLVM SCC pass
- Extended functionality of FunctionPass
- Implementation of stack analysis (incl. response model, ...)
- Implementation of compile scripts
- Performance analysis
- Security analysis

3 Tool usage documentation

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 -c 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 pass iterates over all functions that can be found in the input file to protect. We use the pass 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.

When the pass has iterated over all functions, we produce the file `graph.txt` where all edges from the entry point of the program to a sensitive function are contained. This file is required during the stack trace verification. We also dynamically generate the code that is injected into the program. I.e., we update a placeholder for the expected hash of the graph.txt file by the hash we generate from the file. Note that it would also be possible to dynamically generate the stack analysis functions with hardcoded adjacency matrix or list as a performance improvement. However, the security of both concepts remains the same.

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.

Note that the instrumented program requires the valid graph.txt file in the same directory.

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

In this section we conduct several experiments to evaluate the performance of our tool. We first discuss on the time required to protect an existing source file. Then, we try to measure the overhead caused by our protection scheme on the protected binary, in terms of memory use and runtime overhead.

Time related measurements are obtained on a Lenovo ThinkPad X250 laptop with an Intel i5-5200U, 8GB of RAM running Manjaro 17.0.1 and Linux kernel 4.9.

5.1 Protection Time

We used the `time` command to measure the time necessary to run our pass and compile the binary 50 times to get more precise estimations.
We used as input a simple program further used in the memory consumption evaluation, consisting of 3 simple functions and then added other dummy functions. Results are shown in Table 1.

There does not seem to be any major increase in the protection time as the input program grows a little bit larger and as additional sensitive functions are protected. Our pass iterates over the basic blocks of each function of the input source file to inject calls to the functions building the shadow stack. The more functions, the more iterations will be performed, simply injecting function calls does not seem like a computationally intensive task. Protecting more functions just results in additional calls to the `verifyStack` function being injected, which should not really increase the protection time by much.

Note that the functions in our test program are very small and only consist of few statements. As the llvm pass iterates over all statements, the number of statements is essential for the protection time rather than the number of methods. An indicator for the number of statements is the size of the binary file resulting from the uninstrumented program. Using real input source files with

bigger functions could result in an increase in the protection time. Considering the results obtained with the simple source files and the fact that the protection only need to be run once, we believe that protection time should not be an issue.

Number of functions	Binary size (byte)	Sensitive Functions	Protection Time (s)
3	8,776	1	13.329
3	8,776	2	13.361
3	8,776	3	13.163
6	8,856	1	13.709
6	8,856	2	13.980
6	8,856	3	13.959
6	8,856	4	13.733
6	8,856	5	13.853
6	8,856	6	13.670
10	9,016	1	14.418
10	9,016	2	14.191
10	9,016	3	14.258

Table 1. Protection time for different programs

5.2 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.

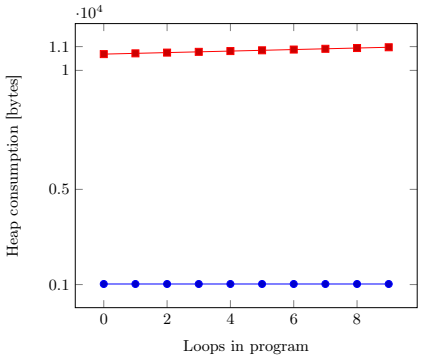

Fig. 1. Used heap for test program

5.3 Runtime overhead

Our defense scheme introduces several potential sources of overhead into the protected application. First, building the shadow stack is done by injecting two function calls respectively at the beginning and end of each function of the program. Then, we need to check the integrity of the call graph file by computing a SHA256 digest to ensure its integrity before parsing it. Finally, when entering a sensitive function, we need to check each pair of functions on our shadow stack against the known call graph.

For our scheme to be useable, we therefore need to ensure that it does not add too much overhead to an existing application. We used a sample application consisting of dummy functions. Using the `time` we measured the time required to run 1,000 times the original application as well as protected versions with an increasing number of sensitive functions. Results are shown in Table 2. In the sample application used, several functions are repeatedly called. This can be considered a worst-case scenario, because our shadow stack will constantly be pushed and poped. In addition, if such functions are listed as sensitive, we will repeatedly check the stack frame when entering them. This observation for instance probably explains the significant increase in runtime from the application with two sensitive functions (line four in the table) to the application with three sensitive function (line five).

Simply injecting the calls to the function responsible for building the shadow stack causes a runtime increase of about 48%. Protecting frequently called functions might also cause a significant increase in the application runtime, whereas for the ones only occasionnally called however it seems to only marginally affect runtime, for instance with five and six sensitive functions (last two lines of the table).

6 Security Evaluation

The defense mechanism we implemented essentially relies on a shadow stack built during runtime and which is further checked against a known call graph. The protection scheme has two main weaknesses that can be tricked by an attacker.

Sensitive Functions	Runtime (s)	Increase
0 (original app.)	0.531	1
0	0.787	1.48
1	1.027	1.93
2	1.167	2.20
3	1.474	2.77
4	1.385	2.61
5	1.725	3.25
6	1.732	3.26

Table 2. Runtime overhead with an increasing number of sensitive functions

First, the attacker might want to manipulate the shadow stack, second she might try to modify the call graph.

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. For example, it is possible to replace the function name with another one. This would allow the attacker to replace a legitimate function call with an illegitimate one. Also, the attacker could remove the push/pop operation and thus add a function call in the graph which would not be available on our shadow stack.

Hence, the question arises if the solution built on a shadow stack is more vulnerable than other solutions. An obvious alternative to the shadow stack is accessing the real stack with the `backtrace` syscall and using this information. However, an attacker can prevent us from calling the syscall or can hook it and give us wrong information. Hence, we cannot provide any security guarantees with this scheme either.

A second vulnerability is the call graph which is written in a plain text file. It is necessary to proof that this file is not corrupted. We therefore calculate a SHA256 digest over the file on instrumentation, add the digest to the instrumented program and check if the checksum is valid before we read from the file.

If an attacker gets access to this digest in the binary, she can defeat the whole scheme by modifying the reference call graph as she pleases. A simple way to defeat this kind of attack is applying an additional layer of obfuscation to this part of the program. This would make reverse engineering more complicated. Also, it is possible to calculate an HMAC over the graph.txt file instead of only a simple digest. To avoid storing the key in plain text in the binary, it is possible to apply white-box cryptography.

The protected binary generated is vulnerable to various static and dynamic patching attacks. Therefore, the use of additional protection techniques is essential to harden the approach against these attacks. E.g., it is possible to apply self-checksumming to protect the integrity of the binary. However, this can be circumvented by known attacks. We therefore propose to extract the stack tracing algorithm from the binary and run it in a secure hypervisor instead. The

hypervisor could be attested with hardware support e.g. using TPM and its remote attestation support.

A summary of the attacks and possible corresponding countermeasures is shown in Figure 2.

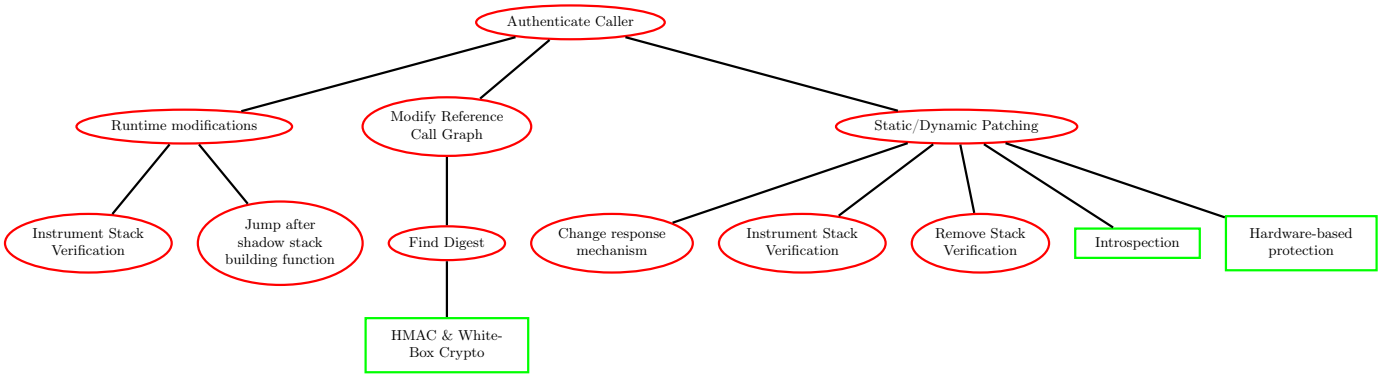

Fig. 2. Attack tree against our stack trace verification scheme

7 Automated attacks

We discussed in the previous section several vulnerabilities of our protection scheme. In this section we present some actual automated attack implementations.

7.1 Bypassing the integrity check

In the directory `/instrumenter/attack` of our docker image, we provide a script that can automatically run one of the attacks we already discussed above.

It is able to replace the original `graph.txt`-file with a new one and automatically patches the correct checksum into the binary. E.g. to modify the binary built from the `memoryTrace3.c` file, run the following commands:

```
cd /instrumenter/build
./compile.sh ../memoryTrace3.c ./memoryTrace sensitiveList.txt
# now, copy the graph.txt file to newGraph.txt and edit the new file
# (e.g. change the number of functions and edges and add a new edge)
# (note that you should adapt the first two lines of the new graph file accordingly:
# the first line is the number of vertices, the second is the number of edges)
cd ../attack
./modifyGraph.sh ../build/memoryTrace ../build/newGraph.txt ../build/graph.txt
cd ../build
./memoryTrace 2
```

Now, the binary `memoryTrace` was run with the new, extended graph.

7.2 Bypassing the stack verification

In the same `/instrumenter/attack` directory, we also provide a script to patch the protected binary and completely bypass the stack verification. It simply inserts a sequence of `NOP` instructions in place of the call that performs the actual verification of the stack.

It expects as argument the name of the binary to patch and will produce a new binary by appending "_nop" to the original binary's name. It can be run as follows:

```
cd /instrumenter/build
./compile.sh ../memoryTrace3.c ./memoryTrace sensitiveList.txt
python ../attack/patchStackVerif.py ./memoryTrace
chmod +x ./memoryTrace_nop
./memoryTrace_nop 1
```

The stack is not checked in the binary `memoryTrace_nop`.

8 Conclusion

Our work is motivated by the search for a purely software-based and highly precise stack verification technique to protect sensitive functions from illegitimate calls. We presented a technique that can instrument any C program to perform stack verification on runtime. Experimental results showed that our scheme requires low overhead of runtime and reasonable overhead of memory consumption. Also, we can guarantee that the scheme introduces no false positives or false negatives. This is encouraging and indicates the high practical relevance of our approach.

However, our security analysis revealed a couple vulnerabilities of the technique. We also presented (automated) attacks to exploit them. As these attacks mainly rely on static/dynamic analysis and consequently patching of the resulting binary, additional techniques are required to protect the binary and the control flow from modifications. Recent research proposes self-checksumming techniques, obfuscation or shielded execution to effectively protect binaries from modifications.