BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATIC SECCOMP SYSCALL POLICY GENERATOR

AUTOMATICKÝ GENERÁTOR POLITIKY SYSTÉMOVÉHO VOLÁNÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	**MAREK TAMAŠKOVIČ**
SUPERVISOR VEDOUCÍ PRÁCE	**Ing. LENKA TUROŇOVÁ**

BRNO 2018

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems Academic year 2017/2018

Bachelor's Thesis Specification

For:	**Tamaškovič Marek**
Branch of study:	Information Technology
Title:	**Automatic Seccomp Syscall Policy Generator**
Category:	Algorithms and Data Structures

Instructions for project work:

1. Study the fundamentals of Linux system calls, tools for monitoring syscalls, Berkeley packet filter, and libseccomp. Conduct research on an intermediate representation of syscalls and optimizer of the intermediate representation.
2. Based on the research, design the intermediate representation and provide a design of appropriate optimizer for it.
3. Implement tool which reads output of strace comment and translates it to the intermediate representation. Implement designed optimizer and translator which transforms optimized structure to a seccomp policy.
4. Evaluate implementation of this tool on selected complex programs.

Basic references:

- Paul Moore. Libseccomp. https://github.com/seccomp/libseccomp, 2012. [Online; accessed 2017-11-02]

Requirements for the first semester:

Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Turoňová Lenka, Ing.**, DITS FIT BUT
Beginning of work:	November 1, 2017
Date of delivery:	May 16, 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

Petr Hanáček
Associate Professor and Head of Department

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

seccomp, libseccomp, strace, optimizer, clustering, C++, policy generator

Kľúčové slová

seccomp, libseccomp, strace, optimalizátor, zhlukovaniem C++, generátor politík

Reference

TAMAŠKOVIČ, Marek. *Automatic Seccomp Syscall Policy Generator*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lenka Turoňová

Automatic Seccomp Syscall Policy Generator

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ms. Ing. Lenka Turoňová (FIT BUT) and by Mr. Bc. Daniek Kopecek (Red Hat, Inc.). The supplementary information was provided by Security Engineering team. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Marek Tamaškovič
July 3, 2018

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

1 Introduction **3**

2 System Calls and Monitoring Tools **4**
- 2.1 System Calls . . . 4
- 2.2 Monitoring . . . 5

3 Security Facilities in Linux **9**
- 3.1 Systrace . . . 9
- 3.2 Seccomp . . . 9
- 3.3 Berkeley Packet Filter and Seccomp . . . 10
- 3.4 Libseccomp . . . 11

4 Solution Design **12**
- 4.1 Requirements . . . 12
- 4.2 Architecture . . . 13
  - 4.2.1 Parser . . . 14
  - 4.2.2 Intermediate Data Structure . . . 14
  - 4.2.3 Optimizer . . . 15
  - 4.2.4 Pitfalls of Clustering . . . 18
- 4.3 Parsing Expression Grammar . . . 19

5 Development of strace2seccomp **20**
- 5.1 Input . . . 20
- 5.2 Output . . . 21
- 5.3 Class Hierarchy . . . 21
- 5.4 Used software . . . 25
  - 5.4.1 Compilers . . . 25
  - 5.4.2 Dynamic and Static Code Analysis . . . 25
  - 5.4.3 Miscellaneous . . . 26
- 5.5 Usage . . . 27

6 Software Verification **29**
- 6.1 Module Testing . . . 29
  - 6.1.1 Params Testing . . . 29
  - 6.1.2 StraceParser Testing . . . 29
- 6.2 System and Acceptance Testing . . . 32
  - 6.2.1 Testing on real programs . . . 32
  - 6.2.2 Test Preparation . . . 33

6.2.3 Test Requirements 33
6.2.4 Results 33

7 Conclusion **36**

Bibliography **37**

A Comparison of libseccomp and raw BPF filtering **40**
A.1 BPF 40
A.2 libseccomp 41

B Output of strace2seccomp **42**
B.1 Example Output no.1 42

Chapter 1

Introduction

Nowadays, when malicious code or malware is becoming more and more sophisticated and pressing security risk, it is really needed to control a program behaviour and monitor what the program is doing in a system. Monitoring program behaviour can be done in many ways and one of the easiest ways is to use Intrusion Detection System (IDS). IDS is an out-of-the-box solution which can monitor i.e. where program wrote or read something and it is not allowed. After that, IDS is reporting this violation.

Another way is to monitor and block system calls (syscalls). Monitoring is performed using tools mentioned in the next chapter. The actual blocking can be performed with mandatory access control (MAC) (Apparmor, SELinux), sandboxing (seccomp) or others mechanisms. MAC refers to a type of access control by which the operating system constraints the ability of a subject or initiator to access or generally perform some sort of operation on an object or target. Seccomp is a Linux kernel module which allows a process one-way transition to secure a state where the process can only use four syscalls. When the process tries to call another syscall then one of the four member's sets is terminated with SIGKILL. The set of allowed system calls can be extended using seccomp-bpf. This extension allows filtering system calls using a configurable policy implemented with Berkley Packet Filter (BPF) rules. This last part is an area on which I would like to focus in my thesis.

I aim to design and develop a tool which helps developers using libseccomp and seccomp-bpf. I plan to create policies for a specific program in a format readable by libseccomp or seccomp-bpf.

Chapter 2 describes syscalls and how to monitor them. In the chapter Chapter 3 of the thesis, I will illustrate how security facilities in Linux, such as systrace and seccomp, work. After the theoretical part, the design and development of a tool will follow. In conclusion, methodology how this tool was tested is described.

Chapter 2

System Calls and Monitoring Tools

In this chapter, I will describe the term system call and make an overview of tools which can monitor the system calls. We will focus in detail on the strace tool which will be used as an input to my tool. The other applications are described briefly not as detailed as the strace tool.

2.1 System Calls

In computer terminology, the term syscall is a way in which a computer program requests a service of the operating system on which is executed on. In other words, system calls are functions used in the kernel itself. The system call appears to a standard developer as a C function call. This design is typical for monolithic kernels. We can find them on every UNIX system. The system call can be called on Linux/i86 multiple ways. One of them is to call interruption no. `0x80` with value of syscall in register `eax`. The seccond and third one is by calling system calls `syscall()` or `sysenter()` and these syscalls are handled by the kernel in a privileged mode. When a user invokes a system call, an execution flow is as follows:

- Each syscall is vectored through a stub in libc. Some syscalls are more complicated than others because of a variable length of the arguments, but the entry point and the end point of syscall are still the same.
- In libc, the number of the syscall is then set to an `eax` register, and the stack frame is also set up.
- An interrupt number `0x80` is called and transferred to the kernel entry point. The entry point is the same for every system call.
- In the table of interrupts, a pointer to interruption handler is found. After that, the execution of the interrupt handler follows which stores the content of the CPU registers and checks if a valid syscall is called.
- The handler finds the corresponding offset in the table of interrupts `_sys_call_table`, where a pointer to the syscall service is stored.
- Control is transferred to the syscall service.
- Syscall returns a value to the register `EAX` on a 32-bit architecture or `RAX` on a 64-bit architecture.

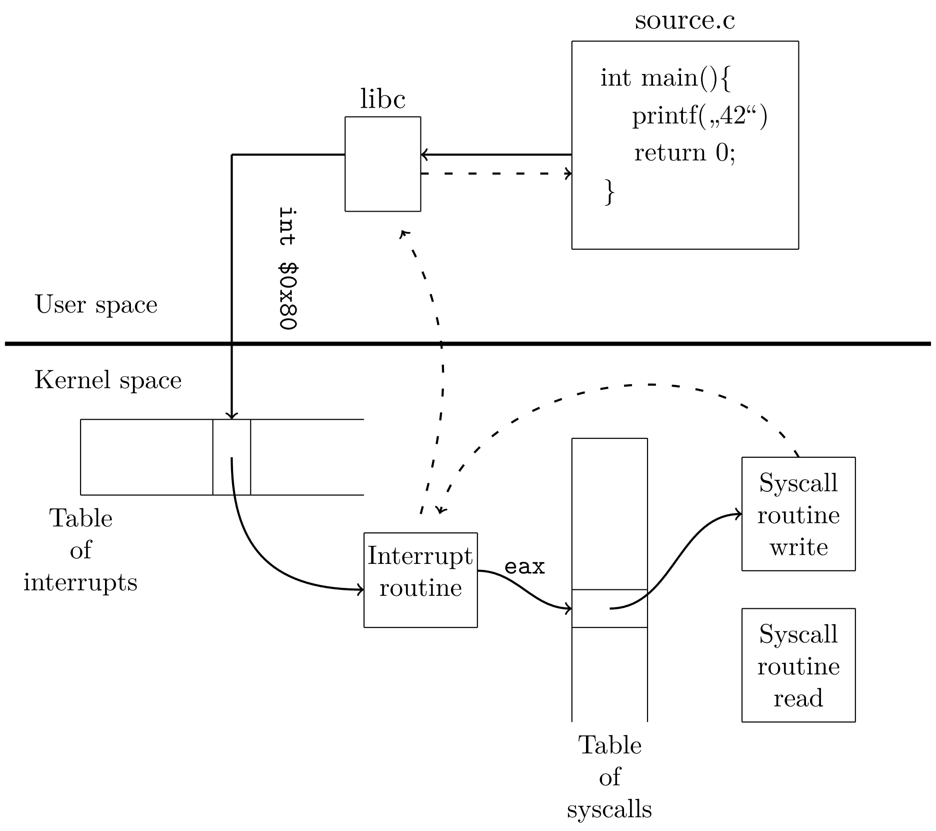


Figure 2.1: Interruption handling in Linux

- At the end of the syscall, `_ret_from_sys_call` is called. This call is done before returning to userspace. It checks if the scheduler should be run, and if so, it calls it.
- Immediately after return from the system call to interrupt handler, `syscallX()` macro checks for a negative return value from the syscall, if so it puts a positive copy of a return value to a global variable `_errno`, for accessing from code like `perror()`.

This procedure is illustrated in Figure 2.1. [25]

2.2 Monitoring

The most used and conventional method for monitoring is tracing, in other words watching what a program is doing during the execution. Tracing involves a specialized logging to record information, useful for debugging, about a program's execution. This can be done in multiple layers, from tracing which lines in the program was executed to individual instructions run on a CPU. Collecting this information can be done with various tools, e.g., strace, ftrace etc.

Strace Strace [6] is a easy to use diagnostic, instructional and debugging tool. You can monitor every syscall or signal made by the program you are tracking. Using this tool it is possible to log what the observed program demanded from the kernel. The individual recorded operations can be, e.g., an attempt to open a file or delete a content of CPU caches.

This tool also shows arguments for the called syscall and it can show data strcutures with their elements, etc. The developer can perform a fault injection for the specified set of syscalls as well, to simulate the program in faulty test cases. Another feature is that the Strace can trace child processes of the observing program. The log on the output will contain the system calls from the primary process and its child processes.

The main advantage of Strace tool is that it does not need any source codes. The observing program has not to be compiled with extra flags nor object files. Also, it does not matter if the application is statically or dynamically linked. This is useful because we only need to execute the binary. These features are helpful for my tool, but this will be more described in a later chapter. The usage of strace tool is straightforward, i.e., when one wants to run ls with strace he types in a command line:

```
>$ strace ls
```

In this case, strace executes the `ls` command, and on the output, it shows which system calls were called. An example of the strace output is in the next figure.

```
execve(„/usr/bin/ls", [„ls"], 0x7ffd0cf4ba60 /* 59 vars */) = 0
open(„/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, { st_mode=S_IFREG|0644, st_size=202163, ...}) = 0
mmap(NULL, 202163, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd781293000
close(3)
```

Strace grammar Strace produces structured output. Simplified version in extended Backus–Naur form (eBNF) [13] you can see in Figure 2.2.

$$\begin{array}{rcl}
\langle\text{grammar}\rangle & \models & \langle\text{system\_call}\rangle \;|\; \langle\text{signal}\rangle \;|\; \langle\text{exit\_line}\rangle \\
\langle\text{system\_call}\rangle & \models & \langle\text{sc\_name}\rangle\ \texttt{"("}\ \{\langle\text{argument}\rangle\}\ \texttt{")"}\ \texttt{"="}\ \langle\text{digit}\rangle \\
\langle\text{signal}\rangle & \models & \texttt{"+++ killed with"}\ \langle\text{signal\_name}\rangle\ \texttt{"+++"} \\
\langle\text{exit\_line}\rangle & \models & \texttt{"+++ exited with"}\ \langle\text{digit}\rangle\ \texttt{"+++"}
\end{array}$$

Figure 2.2: Strace output grammar in eBNF

As you can see the grammar is composed of main parts that are *system_call*, *signal* and *exit_line*. We are interested the first one (*system_call*). The system call is composed of a name of syscall, arguments and return code. The argument can occur in a sequence and it is considered of atomic types (value, constant, structure, array, string, address and there can be find comments as well). The string is represented in a program as a place in memory but strace can dereference this address and show it in analysis.

Ptrace Strace is using ptrace [5] system call. Ptrace is used to implement debuggers and other tools for process monitoring. Basically, the strace call ptrace and attach to a tracee (monitored process). When the connection is established the tracee is halted before and after syscall. Now the tracer (strace) can observe and control the execution as well as inspect memory and registers of (tracee). With this information, strace can determine which syscall was called. During the second halt after syscall, the strace can get information of return value from syscall.

Ftrace Ftrace [4] is an internal tracer which traces events in the kernel. It is designed for developers to examine kernel events. The main feature of this tool is to measure latencies and find issues that take place outside of the user-space. Ftrace is typically considered as a function tracker, but it is a framework of several different tracing utilities. One neat feature of ftrace is measurement of latency among interrupts, the lag between the time when the task is woken up and time when the task is scheduled in. Another frequent use of ftrace is event tracing. In the kernel, there is a massive amount of static event points that can be enabled with a tracefs file system. The event points provide an interface to observe what is going on in the various parts of the kernel.

Dtrace DTrace [3, 17] (shortcut for Dynamic Tracing) is a performance analysis and a troubleshooting tool. It is included in various operating systems, such as FreeBSD, Mac OS, Solaris and Linux. This tool instruments all software, not just user-level software but also operating system kernel and device drivers. It supports dynamic tracing which means dynamically patching while running instructions with an instrumentation code. Static tracing is supported as well, but it needs to add tracepoints to the code. DTrace provides a scripting language called 'D' for writing scripts and one-liners. It is similar to C with AWK elements. With this script, you can create filters and summarize data in the kernel before passing to user-space. This design can decrease the overhead in performance of sensitive systems.

For our purposes, DTrace is too complicated to setup or gather the information about syscalls. You need to write some scripts to define which syscalls you want to be informed with and in our use case, we need every system call.

SystemTap SystemTap [8] is a tracing and probing tool that allows to gather information from probes injected into the kernel. It is similar to Dtrace. It started as a clone of Dtrace because it has uncompatible licence for using in GNU Linux. One of the common thing with Dtrace is that both tools use some type of scripting language. In this case it is named SystemTap. With this language you can specify what happens when some event occurs in the kernel.

SystemTap works as daemon which communicates with a stap program. Stap is a small program that translates the SystemTap script to a kernel module. It is done in a few steps. At first it runs semantic analysis on the script. After that, stap tries to translate it into a C code. The next step is to compile it as a kernel module and load it into the kernel. After load it is working and doing the useful part. When you send a signal to terminate the stap program it will unload the kernel module and stop working.

Similar as Dtrace the SystemTap is too complicated to work as system call monitor and it is not flawless. The Systemtap can not dereference the pointer address in the system call but the strace tool can.

Autrace The Linux Auditing System helps system administrators to create an audit trail. Every action on workstation or server is logged into a log file. This tool can track security-relevant events, record the events and detect misuses or unauthorised activities by inspecting the audit log. You can also set which actions should or should not be reported.

Audit System is composed of two main parts. The first one *autrace* is a kernel component which intercepts system calls, records events and sends these audit messages to the next part. The second component is an audit daemon working in user space. This part is collecting the information emitted by a kernel component. Emitted data is then stored as entries in a log file. As you can see this tool is not for monitoring one specific program, but it is designed to monitor the whole system. In the output, there is specified who and when executed the syscall, current working directory, uid, gid, etc. Above specified functionality is useful for server administrators but not for our work.

Entry example in a log file:

```
type=SYSCALL msg=audit(1434371271.277:135496):  arch=c000003e syscall=2
success=yes exit=3 a0=7fff0054e929 a1=0 a2=1fffffffffff0000 a3=7fff0054c390
items=1 ppid=6265pid=6266 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts0 ses=113 comm=„cat"
exe=„/usr/bin/cat" key=„sshconfigchange"
```

Chapter 3

Security Facilities in Linux

This chapter describes security facilities in GNU Linux operating system. First we mention an old tool named Systrace [21]. Later we will mention a secure computing module named Seccomp [19]. The next topic will be Berkley Packet Filter (BPF) because it is used in seccomp-bpf. The seccomp-bpf is an extension to basic seccomp. This extension can better describe the behavior of seccomp. In the end, there will be a description of libseccomp which is an easy to use library to the kernel syscall filtering.

3.1 Systrace

Systrace is security facility which limits an application's access to the system. It is similar to a newer tool named seccomp-bpf which will be described later. The restrictions of a program are provided via system call blocking. The policy is generated interactively. Operations not covered by the defined policy raise an alarm. When an alarm is raised the user can refine the current policy. Systrace provides an option to generate policies automatically which can be immediately used in sandboxing (Sandbox is a security mechanism for separating programs, usually in order to mitigate system failures or software vulnerabilities from spreading.) [1]. It is not flawless, so it sometimes needs minimal manual post-processing.

This tool provides cybersecurity by providing intrusion prevention. One of the uses is that you run systrace on the server. The systrace monitors all running daemons (daemon is a computer program that runs as a background process, executed on system start up) and can generate a warning when some incident occurs. These alerts can be sent to a system administrator and can provide information what happened.

3.2 Seccomp

A large number of syscalls are exposed to user space of a process. Many of this syscalls are unused for the entire lifetime of the process. This exposes a possibility to misuse some syscalls to corrupt the process itself. A particular subset of applications benefits from a reduced set of syscalls by reducing exposed kernel surface to process. The filtering is done by seccomp. Seccomp filtering provides means for a process to reduce the set of syscalls available to the process [9].

In most contemporary distribution, a kernel module named Seccomp [19] is enabled. Sec-comp stands for the shortcut of Secure Computing Mode. This module provides one

[1]https://www.wikiwand.com/en/Sandbox_(computer_security)

way transition to a secure mode which restricts a thread to a few system calls `read()`, `write()`, `exit()`, `sigreturn()`. If the thread tries to call another system call then the one from the four-member set, the whole process is terminated with signal `SIGTERM`. The drawback of this solution is that these four system calls are not enough for application to run correctly.

3.3 Berkeley Packet Filter and Seccomp

The seccomp filter mode allows developers to write BPF programs that determine if a given syscall will be allowed or not. That allowance can be based on a system call number or specific syscall argument values. Only the passed values are available, as any pointer are not dereferenced by the BPF. Filters can be installed using `seccomp()` or `prctl()`. First, the BPF program should be constructed, then installed in the kernel. After that, every system call triggers the filter code. The installed filter cannot be removed or modified. Another property of applied filter is that the every child process inherits the filter from a parent process when using `fork(2)` or `exec(2)`.

A BPF language came in 1992 for a program called tcpdump which is a monitoring tool for network packets. The volume of packet can be colossal, so it makes the transfer to user-space expensive. The BPF provides a way to do filtering in the kernel and the user space only handles those packets which is interested in.

The seccomp filter developers realised that they wanted a very similar task. After that, the BPF was generalized to allow system call filtering. After the update, there is a tiny BPF virtual machine in the kernel space that interprets the BPF instructions.

The next update of BPF was eBPF which stands for extended BPF. This update was released in Linux Kernel 3.18 for tracepoints later in 3.19 for raw sockets and in 4.1 for performance monitors. The eBPF brought the performance improvements and new capabilities.

The eBPF virtual machine is widely used in the kernel for various filtering:

- eXpress Data Path (XDP) *is a high performance, programmable network data path in the Linux Kernel*
- Traffic control,
- Sockets,
- Firewalling $xpf\_bpfmodule$,
- Tracing,
- Tracepoints,
- kprobe *dynamic tracing of a kernel function call*,
- cgroups.

eBPF - Specification The eBPF virtual machine has a 64-bit RISC architecture designed for one to one mapping to 64-bit CPUs. Instructions are similar to classic BPF for simple conversion to eBPF. The old format had registers A and X instead of current 11 registers grouped by function as described below [24].

- R0 exit value for eBPF
- R1 - R5 function call arguments to in-kernel functions
- R6 - R9 callee-saved registers preserved by in-kernel functions
- R10 stack frame pointer (read only)

So far 87 internal BPF instructions were implemented. Opcode field has a room for new instructions. Some of them may use 16/24/32 byte encoding.

Same as the original BPF (the new format runs within controlled environment) is deterministic and the kernel can easily prove that. The safety of a program can be verified in two steps. First step does depth-first-search to forbid loops and Control flow graph (CFG) validations. The second step starts from first instruction and descends all possible paths in CFG. It simulates execution of every instruction and examines the state of registers and a stack [24].

eBPF - Instruction Encoding. An eBPF program is a sequence of 64-bit instructions. All eBPF instructions use the same design of instruction encoding which is shown in Figure 3.1. As you can see in the figure, there are 5 parts that are opcode (operation code), dst (destination), src (source), offset, immediate [24].

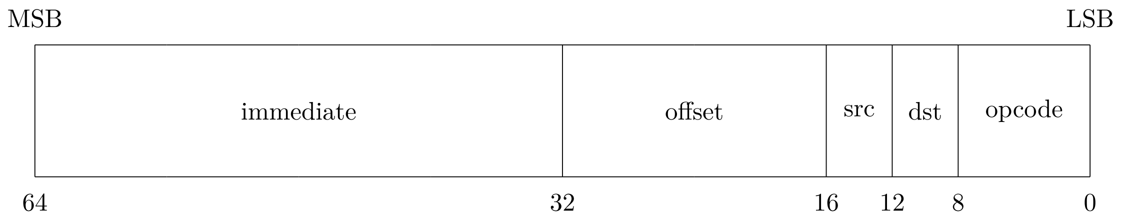


Figure 3.1: eBPF instruction encoding

3.4 Libseccomp

Libseccomp [20] is easy to use library which provides a platform-independent interface to the Linux Kernel's syscall filtering. The libseccomp API is designed to abstract an user from underlying BPF based syscall filtering and present a more conventional function-call based filtering interface. The interface should be more familiar to and quickly adopted by application developers. The comparison of libseccomp and raw BPF filter is shown in Appendix A.1 and A.2.

The library accept on input a set of rules which are later transformed into a BPF format used in seccomp. One of the advantages of a libseccomp is that you can write a function call based filter. This filter is then translated to BPF and after that it is loaded into seccomp as filter. This method is not transitive from function call filter to BPF. There are some differences but they are on so small-scale that they can be ignored.

Another advantages of seccomp is that it has a permissive mode in which every syscall violation is reported to the user. This feature can be helpful if you want to obtain information which syscalls was called. This use case is really similar to the syscall monitoring. But it is really tough to depend on this output because it is in development and is dependent on autrace.

Chapter 4

Solution Design

This chapter will describe the technical part of the thesis. We will discuss requirements and particular parts, its architecture and issues, we have to deal with.

4.1 Requirements

We will require from the application to fulfill the following requirements:

1. Application will have only command line interface (CLI).
2. The application will be implemented in C++17 [12].
3. Application will be designed with consideration of good OOP.
4. Application will consist of these main parts:
   (a) parser
   (b) optimizer
   (c) policy generator
   (d) logger
5. Parser will be implemented using parsing expression grammar (PEG) [7] design.
6. Optimizer will have at least three optimizing methods:
   (a) strict - without use of advanced methods,
   (b) minimax - possibility to count an interval interval between minimum and maximum value found in a set of arguments,
   (c) advanced - combination of above methods.
7. Policy will be generated with libseccomp [20] syntax as C language [14] code.

4.2 Architecture

The architecture of this application is based on architectural patterns. In this case, *Pipe-and-Filter* [15] architectural pattern was used. This pattern best fits our problem. A big inspiration came from compilers. They are very similar to this application. They break down the processing required for input into separate components (or filters), each performing a single task. By normalization the format of the data that each component produces, this component can be arranged as a pipeline. The pattern is suitable for change or adding components and reduces duplicit code. But in this case, this pattern is slightly modified. The data in the pipeline is processed as the whole batch.

The similar components with compilers. They have got a parser, optimizer and output generator as well and every component is dependent on a precendent component. There are two main cases as shown in Figure 4.1.

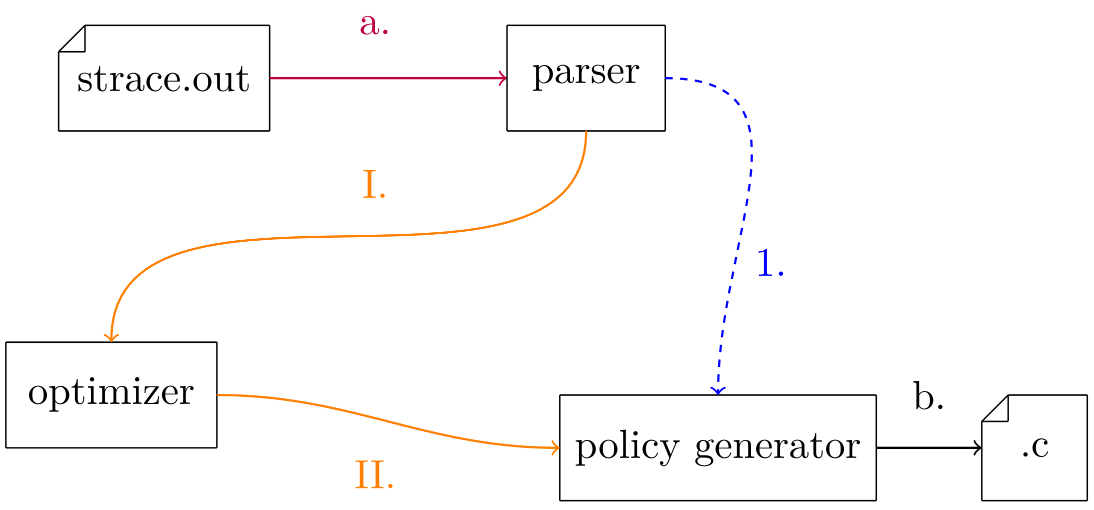


Figure 4.1: Architecture of strace2seccomp

Without Optimizer. In this case, optimizer is not in the pipeline, a raw input is translated into libseccomp commands. This pipeline is shown in Figure 4.1 *(path: a.1.b. in Figure 4.1)*. There are no optimizations when optimizer is turned off.

The main problem of not using a optimizer is that a system call filter does not properly work. There is a possibility that any minor change in system call parameters can results into a program termination. In complex program, there is no way to be definitely sure wheather every syscall was caught. That can be a big problem in programs which use seccomp. That is the main reason why this option is not recommended.

However there are some users which may not want to optimize the strace. The reason for not running the optimizer is that their application does not have variable parameteres in system calls. Every syscall is the same on any running instance of the application. Attentive reader may notice that this option can be used only in small applications which have some limited functionality.

The main disadvantage of this solution is that it is too robust to place it in a code and is very strict. It can kill a program even in a false positive case when an user changes some of the parameters that was not covered in strace input files.

With Optimizer. In this case when optimizer is turned on *(path: a. I. II. b. in Figure 4.1)*, we can specify which type of optimization we want. As mentioned in Section 4.1, there are two variants of optimization. Those variants can be switched with runtime arguments in CLI.

The pitfalls of this case is allowing program to continue even in inappropriate circumstances. Invalid circumstances can be defined as a bad syscall argument treated as a valid. It can happen when you allow a set of arguments for specific syscall. This is not secure however it is more suitable for work than the case without optimizer.

You can minimize these pitfalls by providing a lot of strace input files. The best case is when you provide strace files from every major complex test case (with big prime path coverage).

4.2.1 Parser

The parser is crucial part of the whole application because it will put everything in an IR (intermediate representation). Input to the parser is a output from the `Strace` tool. The output was described in Section 2.2 and correct configuration of strace to generate valid output for `strace2seccomp` will be described in Section **??**. As you can see the output is in a structured form and has an unambiguous syntax which means that no error should occur during the parsing part. When syntax error occurs, the program should inform where the error is located in the input file. Next step should be a proper exit. Parser should have an option to identify all errors in the input file which can be helpful for identifying more errors at once. Another feature is that the parser can print structured parsed data.

4.2.2 Intermediate Data Structure

One of the main part of strace2seccomp is an intermediate data structure (IDS) in which the individual system calls are stored. The main idea of this abstract data type (ADT) is to be simple and readable with smallest redundancy possible. This can be done only with good abstraction and good design.

IDS is represented as a tree structure. In this structure, there are different information about syscall represented at a different level in the tree. The root node has child elements which represent individual system calls. In this case, we call these nodes system call node (SCD). In SCD, information about a system call number is stored and it have multiple children. The n-th level represents the n-th argument of a specific system call. The whole system call (including arguments) can be read as a path from the root node to the leaf node. The IDS representation is shown in Figure 4.2.

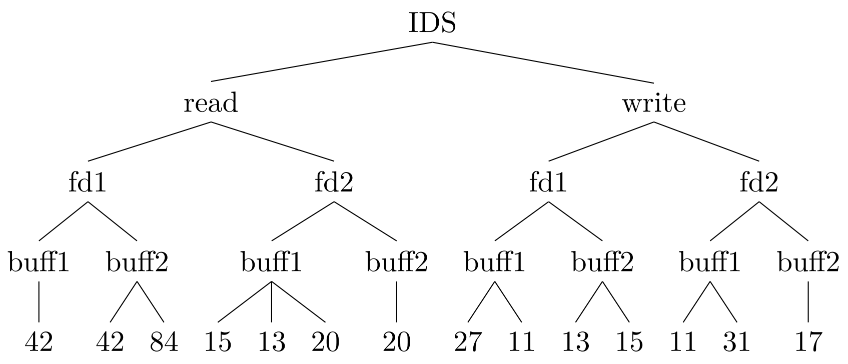


Figure 4.2: Visualized IDS as a tree

4.2.3 Optimizer

Optimizer is the main part of this tool. This part will reduce the intermediate data structure (IDS). There are three approaches to reducing IDS (strict, advanced and minmax).

Strict optimization. Strict optimization is defined as 1:1 to strace input file. It means that it will interpret every record in strace as a strict rule. So only that one case is only possible to run. Every minor change in system call will kill the process.

Algorithm 1: Weak optimization

```
Input: intermediate data structure IDS
Output: list of rules rules
1 foreach syscall sc in the IDS do
2     foreach argument arg in the syscall sc do
3         num_arg ← get_num_args($p$);
4         for lvl ← 0 to num_arg do
5             intervals.append(get_minmax(arg, lvl));
6         end
7         rules.append(sc, intervals);
8     end
9 end
```

Minmax optimization. Minmax optimization is the most basic one of this set of optimizations. The main idea is to find minimum and maximum for each argument of each syscall. The model uses a simple techniques to find extremes on every position of the system call. Firstly it serialize the arguments from n-th position. Then the serialization searches for extremes.

The abstracted algoritm follows as this:

1. Serialize n-th argument position of a system call.
2. Search for extremes in serialization.
3. Increment n.
4. If n is bigger than a number of arguments in syscall then exit.

5. Go to number 1.

Algorithm 2: Weak optimization

```
Input: intermediate data structure IDS
Output: list of rules rules
foreach syscall sc in the IDS do
    foreach argument arg in the syscall sc do
        num_arg ← get_num_args(p);
        for lvl ← 0 to num_arg do
            intervals.append(get_minmax(arg, lvl));
        end
        rules.append(sc, intervals);
    end
end
```

Clustering. Clustering is learning algoritm from a family of unsupervised machine learning. It is a bunch of numerous operations focused on decomposition of informations. When we decompose information then we can clasify it by clasificators. One of them is clustering. Clustering has many definitions, e.g. in book about data mining from Carlo Vercellis [2] is clustering defined as "*Clustering models is to subdivide the records of a dataset into homogeneous groups of observations, called clusters , so that observations belonging to one group are similar to one another and dissimilar from observations included in other groups.*".

The goal of this method is to find subsets (clusters) in given set. Cluster is defined by Paolo S. R. Diniz and group [16] as "*In describing a cluster, most researchers consider internal homogeneity and external separation, i.e., patterns in the same cluster should be similar to each other, while patterns in different clusters should not. Similarities and dissimilarities both should have the potential to be examined in a clear and meaningful way.*".

The clasification of raw data can be done in mulitple ways. Some basic methods are: **[[mention some example algorithms]]**

- partitioning,
- hierarchical,
- density based clustering,
  - DBSCAN clustering,
- fuzzy clustering,
- model-based clustering.

Advanced optimization. Advanced optimization is defined as combination of both strict and weak optimizations. In some specific cases, it will use only the exact values and in other cases, it will use weak optimizations. This combination should be more strict than the weak optimization and weaker than the strict optimization.

In this case, DBSCAN clustering method is used [23]. The model introduced by DBSCAN uses a simple minimum density level estimation. It defines a treshold for the number of neighbors (minPts) within the radius ϵ. Objects with more than treshold neighbors within ϵ are treated as core points. The intention of DBSCAN is to find all areas, which satisfy at least the minimum density separated by areas with lower density (noise). Every point in ϵ radius is a part of the same cluster as a core point. If any neighbour is again a core point, their neighbourhoods are transitively included to a core point. This is very simple and basic algorithm as you can see later in this section. The strength and weakness of DBSCAN clustering is that it does not require a number of output clusters.

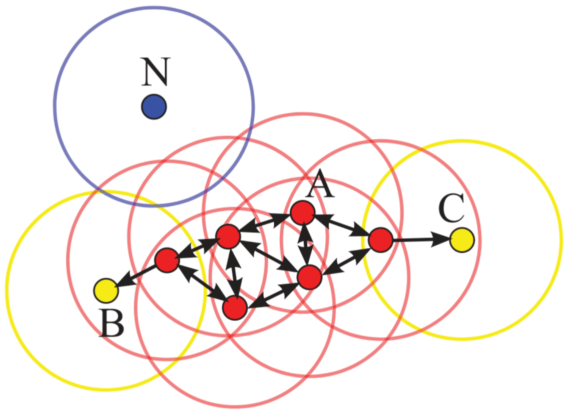


Figure 4.3: Illustration of DBSCAN cluster model

Figure 4.4 illustrates the model DBSCAN. Following parameters are defined:

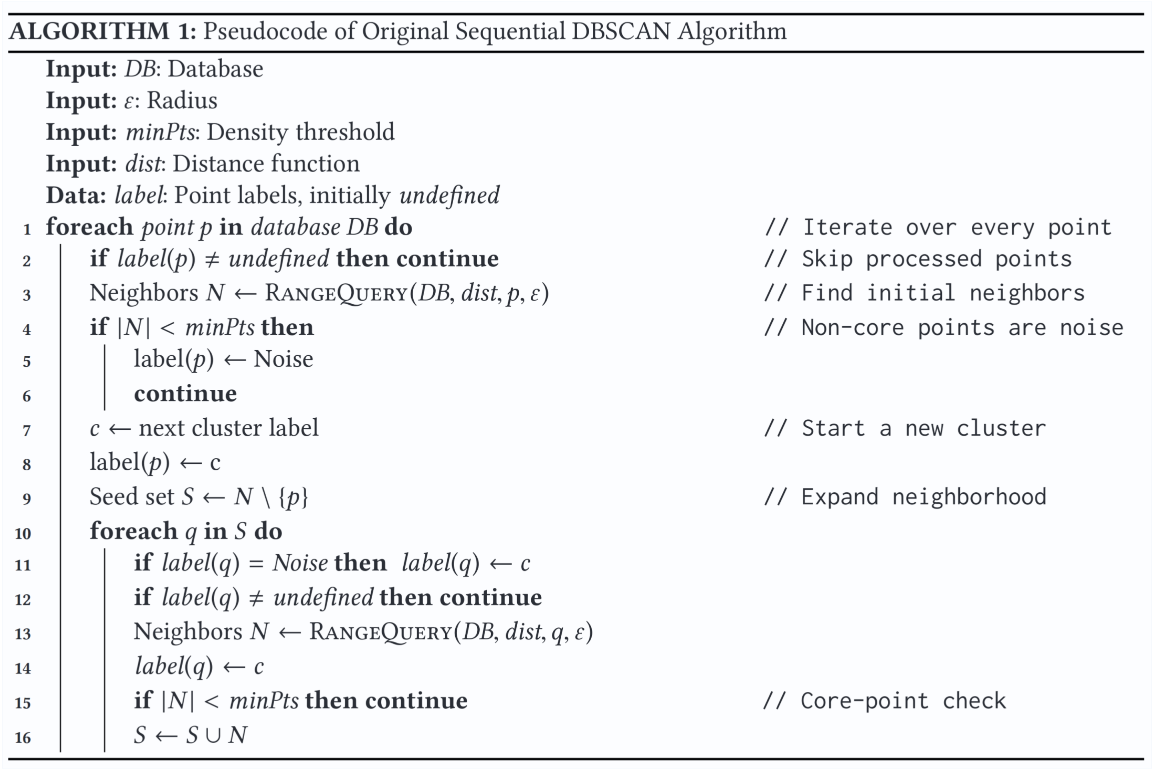
- minPts is 4, and
- *epsilon* is indicated by circles

In this picture you can see multiple points and four of them named A, B, C, N. Arrows indicate direct density reachability. Points A, B, C are density connected and B, C points are border points. N is not density reachable (it is not in any ϵ radius). Any point as N is considered as noise point.

The abstracted algorithm is:

1. Find neighbors in ϵ radius of every point, and find core points with more that minimum neigthnours (minPts).
2. Find connected core points on the graph and merge them into clusters.
3. Assign every non-core point to a core point. If the non-core point is in ϵ radius of that core-point else assign it to noise.

The whole algorithm is:[[rewrite it and add output]]



ALGORITHM 1: Pseudocode of Original Sequential DBSCAN Algorithm

```
Input: DB: Database
Input: $\varepsilon$: Radius
Input: minPts: Density threshold
Input: dist: Distance function
Data: label: Point labels, initially undefined
foreach point p in database DB do                     // Iterate over every point
    if $label(p) \neq undefined$ then continue        // Skip processed points
    Neighbors $N \leftarrow$ RangeQuery(DB, dist, p, $\varepsilon$)   // Find initial neighbors
    if $|N| < minPts$ then                            // Non-core points are noise
        label(p) $\leftarrow$ Noise
        continue
    $c \leftarrow$ next cluster label                 // Start a new cluster
    label(p) $\leftarrow$ c
    Seed set $S \leftarrow N \setminus \{p\}$         // Expand neighborhood
    foreach q in S do
        if $label(q) = Noise$ then $label(q) \leftarrow c$
        if $label(q) \neq undefined$ then continue
        Neighbors $N \leftarrow$ RangeQuery(DB, dist, q, $\varepsilon$)
        $label(q) \leftarrow c$
        if $|N| < minPts$ then continue               // Core-point check
        $S \leftarrow S \cup N$
```

Figure 4.4: Illustration of DBSCAN cluster model

4.2.4 Pitfalls of Clustering

The clustering of the arguments of system call has some pitfalls. Some of them are described bellow.

Address space layout randomization. ASLR was introduced in 2001 by Page EXec (PaX) team to defense over buffer overflow attack, [29, 11]. The goal of ASLR is to provide randomness into address space of a given task. This will add another layer of protection against exploits which uses buffer overflow to change behavior of attacked software.

The idea behind ASLR is that the memory segments (stack, heap, libraries, ...) are located on different offsets during the runtime. With this technique you can achieve that exploits can not know where exactly is located, e.g., a stack segment in the memory. When ASLR is enabled, the analysis of the addresess in a system call make no sense. You can not know on which offset any segment will be.

ASLR is by now implemented in many modern OS (MS Windows, GNU/Linux, NetBSD, OpenBSD, MacOS). In Linux, this feature is enabled by a kernel. You can get information if ASLR is enabled in file e.g.:

```
cat /proc/sys/kernel/randomize_va_space
```

Value description:

- **0** ASLR is turned off,

- **1** ASLR is enabled only for Shared libraries, mmap(), VDSO, stack and heap,
- **2** ASLR is fully enabled.

String represented in memory The strings are represented in memory as an offset where the string begins and ends with null termination. [14] For this reason seccomp can not operate with strings and we will skip string arguments as well.

4.3 Parsing Expression Grammar

Parsing expression grammar was introduced by Ford in 2004. [10] PEG is a type of analytic formal grammar which describes a formal language in terms of a set of rules for recognizing strings in the language. This type of grammar is really similar to a top-down parsing languages. As well it looks very similar to context-free grammars.

Parser that parses the PEG is named a packrat parser. Packrat parser can be easily constructed for any language described by an LL(k) or LR(k) grammar, as well as for many languages that require unlimited lookahead and therefore are not LR. Packrat parsers are also much simpler to construct than bottom-up LR parsers, making it practical to build them by hand. It can directly and efficiently implement common disambiguation rules such as *longest-match*, *followed-by*, and *not-followed-by*, which are difficult to express unambiguously in a context-free grammar or implement in conventional linear-time. Finally, both lexical and hierarchical analysis can be seamlessly integrated into a single unified packrat parser.

The main disadvantage of packrat parsing is memory consumption. The worst case asymptotic computational complexity is very similar to the conventional algorithms (linear in the size of the input).

The one of the many problems with right to left parsing algorithm is that it computes many results that are never needed. Other inconvenience is that we must carefully determine the order in which the results for a particular column are computed. *Packrat parsing* is a lazy derivation of a tabular algorithm that solves both of these problems. It computes results only when they are needed, in the same order as the original recursive descent parser would.

Addition	←	Multiplication '+' Addition | Multiplication
Multiplication	←	Primary '*' Multiplication | Primary
Primary	←	'(' Addition ')' | Number
Number	←	'0' | . . . | '9'

Table 4.1: Example of a grammar for a trivial language

Chapter 5

Development of strace2seccomp

5.1 Input

As I mentioned earlier strace tool was chosen because it is easy to use system call monitoring tool. It can monitor what observed program demanded from kernel. With strace tool it is possible to trace child processes. The main advantage of strace tool is that is does not need any of the source code files, program is not compiled with extra flags or withount any library or it does not have to be statically linked. This were the reasons why I chose the strace tool as an input for the strace2seccomp.

The output from strace tool has to be normalized before procesing with strace2seccomp. The normalization is done by providing a few runtime arguments to the strace tool. Example:

```
$ strace -s 0 -xx -o dataset -ff nautilus
```

I would like to describe what they are doing in the first place.[6]

-s is a string size. We are setting a string size to zero because the libseccomp does not have the ability to work with strings. It does not know how long is the string or if it is really a string. By this option the filenames are not affected. They are printed in full length.

-xx this option will switch the format of strings to a hexadecimal format. It is much easier to parse strings in this format. It affects the filename as well. This option is used because sometimes a non ASCII (UTF–8) character can occure in filename.

-f traces children processes.

-o is used for specifying the output file.

-ff is helpfull when you are tracing a multiprocess program. In this case it will create a multiple files in format NAME.PID where NAME is a provided filename in option `-o` and PID is a process id.

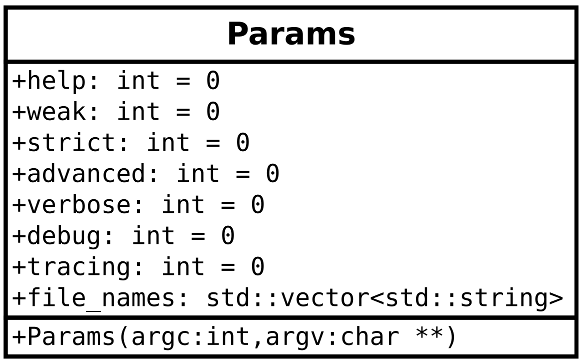
5.2 Output

The strace2seccomp tool is generating a source code for C/C++. In the source code is used seccomp library to provide system call blocking. The example of a generated source code is in Appendix B.1

The source code can be generated in multiple ways. The basic settings are that it will create only function calls with arguments. The extension to this it can add a function prolog and a function epilog. This option is useful when you want to copy and paste the output of strace2seccomp into your source code. The last thing which can be added into output is multithread or multiprocess support. When this option is set up, in the output code will be located part which turns on filter synchronization among threads or processes.

5.3 Class Hierarchy

This section provides information about class hierarchy in this project. The color of classes in figures does not mean anything it is there only for better readability.



```
+help: int = 0
+weak: int = 0
+strict: int = 0
+advanced: int = 0
+verbose: int = 0
+debug: int = 0
+tracing: int = 0
+file_names: std::vector<std::string>
+Params(argc:int,argv:char **)
```

Figure 5.1: Class diagram that shows representation of runtime arguments

At the beginning is the `Params` class as we can see in Figure 5.1. In constructor of `Params` class is executed getopt library to acquire a runtime arguments. The omitted arguments are then saved in class variables. By this variables the whole tool is instrumented.

The next class diagram in Figure 5.2 show us an Intermediate Data Structure (IDS). The IDS is handled with multiple components in this project, i.e., parser saves information there, etc., description is in Section 4.2.2. As you can see the `_argument_t` class has three constructors. Here could be used design pattern factory however there are only two different cases of creating an object so it was not needed. The `_argument_t` class contains individual data about argument, e.g., the format value says if the argument is in 'key=value' or 'value' form. The actual value is stored in container `std::variant`. This is the C++17 equivalent of enumeration in C language.

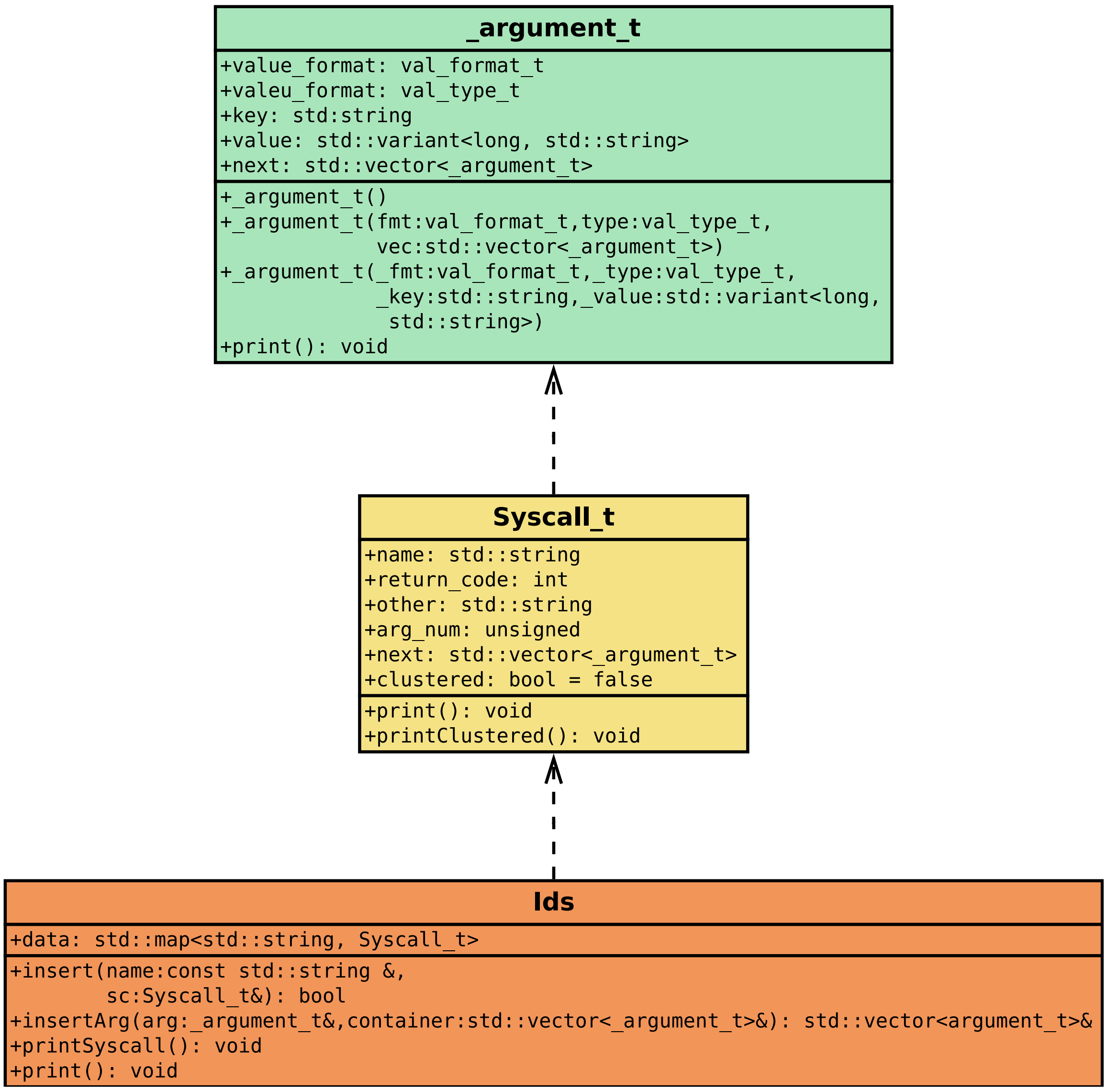

_argument_t
+value_format: val_format_t
+valeu_format: val_type_t
+key: std:string
+value: std::variant<long, std::string>
+next: std::vector<_argument_t>
+_argument_t()
+_argument_t(fmt:val_format_t,type:val_type_t, vec:std::vector<_argument_t>)
+_argument_t(_fmt:val_format_t,_type:val_type_t, _key:std::string,_value:std::variant<long, std::string>)
+print(): void
Syscall_t
+name: std::string
+return_code: int
+other: std::string
+arg_num: unsigned
+next: std::vector<_argument_t>
+clustered: bool = false
+print(): void
+printClustered(): void
Ids
+data: std::map<std::string, Syscall_t>
+insert(name:const std::string &, sc:Syscall_t&): bool
+insertArg(arg:_argument_t&,container:std::vector<_argument_t>&): std::vector<argument_t>&
+printSyscall(): void
+print(): void

Figure 5.2: IDS representation in class diagram

`Syscall_t` class in Figure 5.2 is dependant on the `_argument_t`. This class contains information about syscall, e.g., name, number of arguments, and actual arguments. In this class is defined `print()` method that can print structurised data from syscall.

Class `Ids` in Figure 5.2 describes the whole IDS. In this class is located class variable of type `std::map<std::string, Syscall_t>` wich hold syscall data. This means that `Ids` class is dependant on class `Syscall_t`. The `Ids` class includes methods, e.g., for structuralised print of whole class, or syscall only. It has implemented method for inserting new Syscall object into the map.

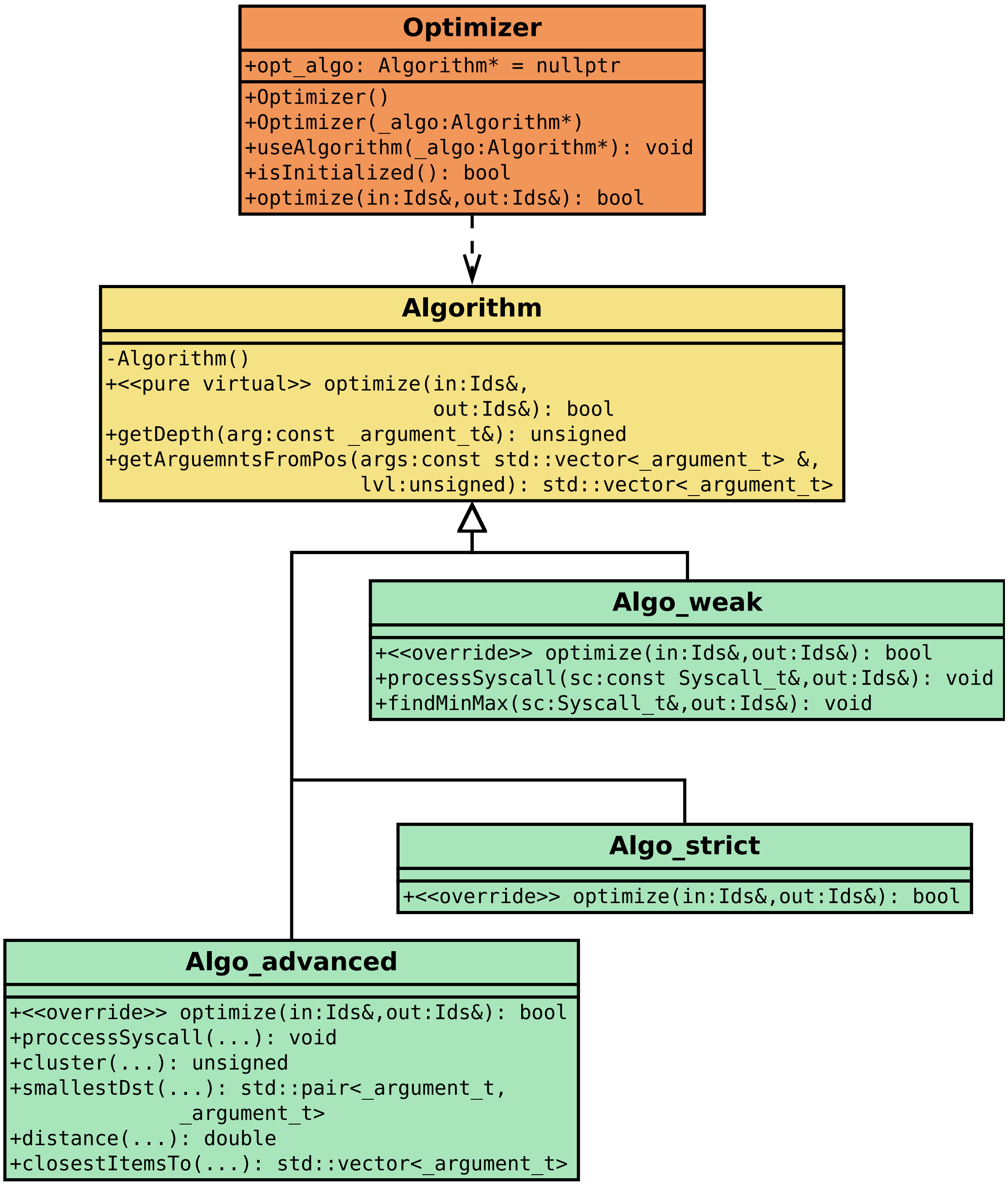

Optimizer
+opt_algo: Algorithm* = nullptr
+Optimizer()
+Optimizer(_algo:Algorithm*)
+useAlgorithm(_algo:Algorithm*): void
+isInitialized(): bool
+optimize(in:Ids&,out:Ids&): bool
Algorithm
-Algorithm()
+<<pure virtual>> optimize(in:Ids&, out:Ids&): bool
+getDepth(arg:const _argument_t&): unsigned
+getArguemntsFromPos(args:const std::vector<_argument_t> &, lvl:unsigned): std::vector<_argument_t>
Algo_weak
+<<override>> optimize(in:Ids&,out:Ids&): bool
+processSyscall(sc:const Syscall_t&,out:Ids&): void
+findMinMax(sc:Syscall_t&,out:Ids&): void
Algo_strict
+<<override>> optimize(in:Ids&,out:Ids&): bool
Algo_advanced
+<<override>> optimize(in:Ids&,out:Ids&): bool
+proccessSyscall(...): void
+cluster(...): unsigned
+smallestDst(...): std::pair<_argument_t, _argument_t>
+distance(...): double
+closestItemsTo(...): std::vector<_argument_t>

Figure 5.3: Class hierarchy that demostrate relationship among algorithms and optimizer

The Figure 5.3 ilustrate dependency and generalization in class hierarchy in optimizer. As you can see the `Optimizer` is dependant on `Algorithm` class. The `Algorithm` class is generalized by specific algorithms, e.g., `Algo_strict` class or `Algo_advanced` class. The optimizer contains a pointer to the `Algorithm`. By this mechanism is really simple to change the algorithm in optimizer and keep the logical structure. It contains methods for initializing and removing algorithm from an object. The algorithms has to override the `optimize` method in `Algorithm` class thus is pure virtual in the base class. Because of that the compiler requests the optimize method in new class that inherits from `Algorithm` class. The generalization is used for simple extension of algorithms in current solution. Every algorithm includes supportive methods for the optimization, e.g., `Algo_weak` has implemented method `findMinMax`.

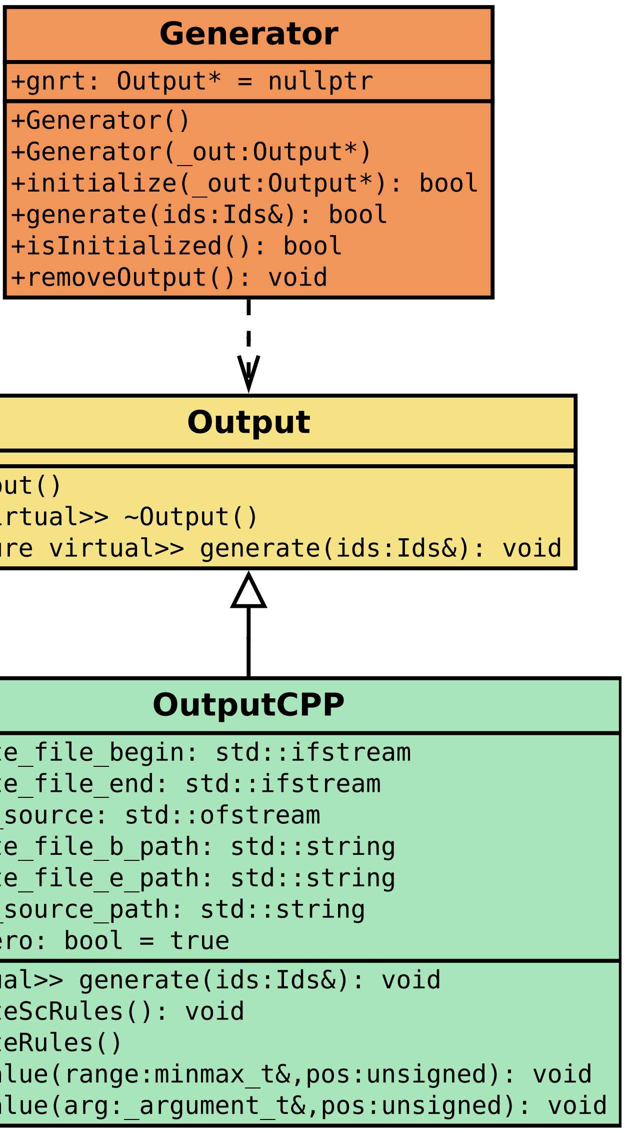

Generator
+gnrt: Output* = nullptr
+Generator()
+Generator(_out:Output*)
+initialize(_out:Output*): bool
+generate(ids:Ids&): bool
+isInitialized(): bool
+removeOutput(): void
Output
+Output()
+<<virtual>> ~Output()
+<<pure virtual>> generate(ids:Ids&): void
OutputCPP
+template_file_begin: std::ifstream
+template_file_end: std::ifstream
+output_source: std::ofstream
+template_file_b_path: std::string
+template_file_e_path: std::string
+output_source_path: std::string
+writeZero: bool = true
+<<virtual>> generate(ids:Ids&): void
+generateScRules(): void
+generateRules()
+writeValue(range:minmax_t&,pos:unsigned): void
+writeValue(arg:_argument_t&,pos:unsigned): void

Figure 5.4: output

In Policy generator is used the same approach as in the optimizer. The hierarchy among classes in policy generator shows the Figure 5.4. **[[finish this after final decision in src]]**

5.4 Used software

In this section, I want to mention which software I used to develop a strace2seccomp tool. Firstly, I mention compilers used in this project and after that valuable tools for developments.

5.4.1 Compilers

In this project, I used two compilers. The reason is simple. Every compiler can detect a different set of errors and warnings during the compilation. And at the time of doing project one of the reason of compiling with two compilers was to compare the execution times with optimizations turned on.

In the project, `libc++` and `libc++ ABI` from LLVM project was used. The reason for using implementation from LLVM project was that in the GNU implementation, there was a bug which affected a C++17 functionality[1] [2]. However, during the later development phase was the bug fixed and both implementation of the library can be used.

GNU Compiler Collection. GNU Compiler Collection is a part of the GNU project. It aims to improve compiler used in the GNU ecosystem. GCC[3] uses an open development environment. It includes front ends for C, C++, Objective-C, Go etc. as well as libraries for these languages. It was firstly written for GNU operating system[4]. The compiler collection is released under the GPL license, other components, e.g., as runtime libraries are distributed under various free licenses.

LLVM/Clang. The goal of Clang[5] project is to provide a new C based language front-end (C, C++, Objective-C,) for the LLVM[6] compiler. It is released under NCSA Open Source Licence. Clang is designed to be highly compatible with GCC. It supports most of the GCC compilation flags and unofficial language extensions[7].

5.4.2 Dynamic and Static Code Analysis

For correct development is necessary to find as many bugs as possible during the this phase. This goal was achieved by ussing dynamic and static analysis tools. For static analysis was used an LLVM project named clang-tidy and for

LLVM/Clang-tidy. Clang-tidy is a clang-based ”linter” tool[8]. Its purpose is to diagnose and fix typical programming errors, like interface misuse, style violation, or bugs that can be deduced via static analysis. Clang-tidy diagnostics are designed to assert code that has

[1] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=877838
[2] https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=258854
[3] https://gcc.gnu.org/
[4] https://www.gnu.org/gnu/thegnuproject.html
[5] https://clang.llvm.org/
[6] http://www.llvm.org/
[7] https://clang.llvm.org/docs/LanguageExtensions.html
[8] http://clang.llvm.org/extra/clang-tidy/

invalid coding standard or is otherwise problematic. It has options to diasable some false positive warnings (e.g. `\\NOLINT`).

AddressSanitizer. AddressSanitizer[9] (ASan) is a memory error detector for C/C++ developed by Google. ASan is very fast and the average slowdown of the instrumented program is ~2x. The tool consists of a runtime library which replaces the `malloc` function and compiler module (currently as LLVM pass). The tool supports multiple architectures, e.g., x86, x86_64, ARM, ARM_64, MIPS, PowerPC64, etc. It is part of the both compilers mentioned above in Subsection 5.4.1.

Usage of ASan is very straightforward. You have to only add compiler arguments:

```
-fsanitize=address -fno-omit-frame-pointer
```

The first parameter turns on the ASan and the second one prints a nicer stack trace in error messages. It is adviced by developers to use optimization, e.g., `-O1`, to get reasonable performance.

5.4.3 Miscellaneous

This section describe miscellaneous software used in the project, e.g., Artistic Style, …

Git Git[10] is a source code manager (SCM). It stands out of the group of SCMs by its branching model. Git allows and encourage you to have multiple local or remote branches that can be entirely independent. This provides features like:

- Context Switching
- Feature Based Workflow
- Role-based Codelines
- Disposable Experimentation

Other benefit of Git is that it does nearly all operations locally. This gives the tool huge speed advantage. Git was built to work with Linux kernel, that means it can effectively handle large repositories. The st2se used repository hosting on GitHub.

Artistic Style Astyle[11] is a source code formatter and beautifuller. Works with C, C++ Objective-C, C# and Java programming languages. The motivation to use this tool is to have uniform code style. Some of the editors by default insert spaces instead of tabs when pressing a key. Other editors have the abillity to insert space before tab lines to ”pretty up” the code (Emacs). The solution to this problem is to use Artistic Style formatter. It can normalise the source code by rules defined in a configuration file provided by a developer of the project.

[9] https://github.com/google/sanitizers/wiki/AddressSanitizer
[10] https://git-scm.com/about
[11] http://astyle.sourceforge.net/

LCOV and GCOV LCOV[12] is an extension to GCOV, a GNU tool which can determine what parts of a program was executed while running particular testcase. It can provide information about how many times that part of program was executed. LCOV implements to GCOV following additional functionality:

- HTML based output with coverage rates indicated by specific color, i.e., green is 100% and red is 0% coverage.
- Support for large projects. It allows you to browse over overview pages that shows coverage data by providing: a directory view, a file view and a source code view.

LCOV was designed like Git to support Linux kernel, but works as well on standard user space applications. This tool uses line coverage technique and it is the poorest coverage from the coverage types point of view.

5.5 Usage

General options		
short format	long format	description
`-h`	`--help`	print this message
`-v`	`--verbose`	turn on verbose mode
`-d`	`--debug`	turn on debug mode
`-t`	`--tracing`	turn on tracing mode
`-A`	`--analyze-grammar`	analyze grammar
`-o=FILE`	`--output=FILE`	set output file

Table 5.1: My caption

Configuration options		
short format	long format	description
`-w`	`--weak`	use weak algotirthm
`-s`	`--strict`	use strict algotirthm
`-a`	`--advanced`	use advanced algotirthm
	`--thread`	generate function prolog
	`--prolog`	add filter synchronization among threads/processes

Table 5.2: My caption

Examples

In Figure 5.5, we can see that verbose mode is turned on and minimax algorithm was chosen for the optimizer. The output of the program will be stored in source.cpp. Files filename1 and filename2 will be used as input.

The command in Figure 5.6 diverges only in the output format. The `-thread` will generate support for multithread or multiprocess applications and `-prolog` switch ensures

[12] `http://ltp.sourceforge.net/coverage/lcov.php`

```
> ./st2se -v -w --output=source.cpp filename1 filename2
```

Figure 5.5: asd

that the filter will be located in function. This behavior is helpful for copy & paste output into an existing program.

```
> ./st2se -w --output=source.cpp filename --thread --prolog
```

Figure 5.6: asd

When we want to check if the grammar in the parser is correct, we can use a built-in tool in parser library. This tool of the parser can be turned on with switch `-A`. Afterwards on standard output will be printed number of found issues. This behaviour we can see in Figure 5.7

```
> ./st2se -A
```

Figure 5.7: asd

Chapter 6

Software Verification

This chapter will describe activities used to assure quality control of the developed tool. First, I want to introduce on which aspects we will focus. One of the aspects is *module testing*. The main purpose of module testing is to detect errors in submodules, in communication among them and in passing data through data structures. Another aspect of verification is *system testing* merged with *acceptance testing*. In this type of testing, we will check if the strace2seccomp tool has a valid architectural design. Next we will check code with *static analysis* tool and get information about errors in code. Static analysis is type of testing which does not require to run the program but requires a source code of the tool. The static analyzer will analyse the source codes with different heuristics and produces a list of detected errors.

6.1 Module Testing

Module testing is a part of the whole quality control process. This testing can provide us how functional are particular components and if they meet the requirements. Table 6.1 shows us the description of module's test suits.

Module / Component	**Test descriptiom**
Params	Validity of recognized runtime arguments
StraceParser	Syntax testing, correct error handling
Output	Check the validity of generated policy

Table 6.1: Test plan

6.1.1 Params Testing

Testing of the parameters was done manually which means by systematically picking border values or some invalid ones and providing them as runtime arguments for the strace2seccomp tool. It was automated with a handmande script.

6.1.2 StraceParser Testing

StraceParser module is responsible for parsing the strace output and translating it into an intermediate data structure. Testing of this module can be done with various techniques. First one which is used is fuzzy testing or fuzzing described in the section below 6.1.2.

Fuzzing

The term fuzzing was first used by professor Barton Miller who used fuzzing to test robustness of UNIX applications in 1989 [28, 18]. Fuzzing is a testing method which generates an unexpected input on tested software and then is observing if the software crashes. The whole process is typically automated or semiautomated which involves repeatedly manipulating and supplying input data to the targeted program. Some modern fuzzers (programs that generates a stochastic input) record every crash or halt of a tested program. The stochastic data are in the most cases invalid to observer thus he can see how application handles invalid states and boundary conditions. The name comes from modern applications tendency to fail due to random input caused by line noise on 'fuzzy' telephon lines.[28, 1, 27] In other literature, fuzzing can be named by these terms:

- Negative testing
- Syntax testing
- Dirty testing
- Rubustness testing
- Protocol mutation
- Fault injection

Fuzzers can be divided into two large groups:

- **Generation-based** fuzzers creates test suite from scratch by modeling the target grammar.
- **Mutation-based** fuzzers needs an (in)valid input file. The file is mutated by various techniques. The mutation can be e.g. bitflip, byte change, duplicate or swap some chunks in the input file. The mutated test case is then provided to a tested program.

For us *Mutation-based* group is interesting because it is easier to setup and is more available in open source community. There are many options to choose so here is a little comparison of most popular fuzzers:

- **Bunny the Fuzzer**[1] is a closed loop, general purpose fuzzer for C programs. The fuzzer uses compiler-level integration which means that it injects reliable instrumentation hooks into an object file. Those hooks enable the fuzzer to trace the program, and can provide real-time feedback. This architecture provides a possibility to considerably improve coverage of testing process. The injection of the hooks needs to be done by the compiler scripts. This fuzzer subset of American Fuzzy Lop. This project is now deprecated.
- **American Fuzzy Lop**[2] is a security oriented fuzzer that add compile-time instrumentation. It is supperset to really similar Bunny the Fuzzer. The fuzzer has implemented many researched fuzzing capabilities (bit / byte flips, simple arithmetics, known integers, test case splicing, ...). Compared to other instrumented fuzzers it

[1] https://code.google.com/archive/p/bunny-the-fuzzer/

[2] http://lcamtuf.coredump.cx/afl/

has moderate overhead and as little configuration as possible. The disadvantage of the tool is that it needs to be executed multiple times to achieve multiprocess or multithread fuzzing.

- **Honggfuzz**[3] is a evolutionary, security oriented, easy to use fuzzer with analysis options. The main advantage of this mutation based fuzzer is that it is multi-process and multi-threded without need to run multiple copies of fuzzer. The file corpus is automatically shared and improved among the processes / threads. Authors of the fuzzer says that it is blazing fast when it works in persistent fuzzing mode'. For monitoring of target (process under test) it uses a low-level interface (e.g ptrace in Linux). It supports several hardware based (Intel BTS, Intel PT) and software based fuzzing methods.

- **Radamsa** is fuzzing tool[4] developed at the Oulu university. The motivation for building this fuzzer was to make robustness testing accessible to independent developers. The existing tools were considered hard to use and to customize to fit the project. The fuzzer is a command line tool, on input it expects multiple files (samples), and generates mutated files. So by the output Radamsa is considered as a mutation-based fuzzer. The tool includes feature aiding in automatizing test runs. This is achieved by specifying number of wanted test runs with unique mutated files. Another parameter that can be set is a random seed for mutation. On the other hand, the tool does not provide a option to monitor target (program under test). Radamsa is a multiplatform so it is built for Windows and Linux.[30]

- **Oss-fuzz** is a complex project developed by Google.[5] The architecture of this project is that the ClusterFuzz (fuzzer tools) automatically pulls the newest source code from repository and it will starts fuzzers and sanitizers [6]. When some bug or fault occurs it is automatically reported to the OSS-fuzz issue tracker. Project owners are then notified with an email about the issue. When the fix is submitted ClusterFuzz automatically verifies the fix, adds a comment to issue tracker and closes the issue.

Fuzzing results

The chosen fuzzer was AFL for its ability of fast deployment and easy of use. The opposite example is Oss-fuzz which is the whole infrastructure for fuzzing and bug hunting.

In fuzzed program, only parsing was enabled every other module was turned off. The fuzzer run straight twenty one days and during this time it was discovered three false positive hangs and no crashes. Fuzzing runs in four threads on Intel Core i7-4810MQ. Overall the fuzzed program was executed 3,567,228,619 times.

The number of execution per second was not stable on master thread because it was necessary to instrument other three processes. This instability can be seen among Figures **??**, **??**, **??** and **??**.

[[check the fuzzer output]]

[3]http://honggfuzz.com/

[4]https://github.com/aoh/radamsa

[5]https://github.com/google/oss-fuzz

[6]Dynamic testing tool that can detect bugs and faults during the execution. Typical sanitizers are *ASan*, *DFSan*, ...

6.2 System and Acceptance Testing

In the book of Software Acceptance Testing, acceptance testing is defined as: "Acceptance testing is the formal testing activity that presents the product to the customer by enterprise (many times it includes stakeholders as well). This activity represents demonstration of a software product and shows to the customer that the requirements fulfilled its obligations. By this activity you can decide if the product is ready for deployment. The software items must be examined to ensure that the provided product is complete i.e. the architecture must be audited to check if it accurate reflects the software configuration. The test results are audited by functional configuration audit to certify that the software product satisfies its specification etc." [22].

Acceptance testing will consist of steps as generate policies, i.e., a whitelist of system calls from strace of given program and applying the whitelist into carefully picked programs. Afterwards, we run the testsuite provided for that program and by this way we can immediately see if the generated whitelist works correctly.

6.2.1 Testing on real programs

For this testing four programs was picked. RedHat Inc. demands one, and that is USBGuard[7]. USBGuard is an open source project developed under GNU GPL v2.0 licence. It implements USB device authorization which can be set up locally or with centralized management [26]. USBGuard has a sophisticated architecture which is complicated enough for use in this testing. Another advantage of USBGuard is that it has large enough testsuite which will be helpful in the evaluation of generated policies by `strace2seccomp`. In this project, we will focus only on a daemon because it is most involved in the whole project.

Next program is called *cp* which is part of a GNU project called *coreutils*[8] . The *cp* is a program used for copying files across filesystem typically on POSIX operating systems. One of the advantages of this project is that it has a relatively big testsuite. The drawback of this program is that it is a part of a big project and is though to acquire strace logs from testsuite. This problem I solved by manually patching the Makefile which was responsible for running tests. The patch will add strace command with appropriate flags before the invocation of *cp*.

Another program is *find* from GNU project called *findutils*[9] . The *find* is a program used to locate files or directories in filesystem. It is specific for GNU Linux as well as cp. This program has good testsuite as a *coreutils* project does. The drawback of the project complexity is similar to *coreutils*. In *find* is a high number of system calls that differ only in provided arguments thus it will test the robustness of clustering in the `strace2seccomp` tool.

The last program is a school project from *Network Applications* class called *testovac*[10] . This project was chosen for its multithreaded architecture. With this program, we can test if the filter is successfully distributed on other threads. The *testovac* does not have a good testsuite thus we only run it in one scenario. However, the point of this test is to see if the provided seccomp filter works on multithreaded applications.

[7] https://usbguard.github.io/

[8] https://www.gnu.org/software/coreutils/coreutils.html

[9] https://www.gnu.org/software/findutils/

[10] https://github.com/tammar96/ISA-testovac

6.2.2 Test Preparation

For the testing, we need to set up an environment. For this purpose, we have a script which will download software on which will be stare2seccomp tested. The script is named `./setEnv` and is located in testsuite folder. The script has multiple subcommands:

- `prep` will download, extract and configure tested programs (*coreutils*, *findutils*, USB-Guard, *testovac*)
- `applyPolicy` will patch source codes with generated policies
- `make` will run make for all tested programs
- `tests` will run testsuits
- `clean` will clean the testing folder
- `revertPatches` will revertPatches
- `straceON` will turn on strace output while runing testsuite
- `straceOFF` will turn off strace output while runing testsuite

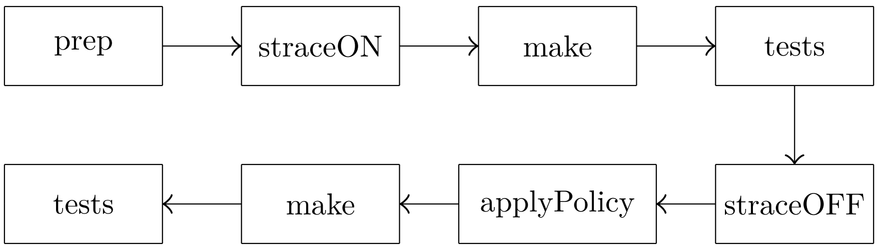


Figure 6.1: Steps to set environment, gather strace logs, make binaries, run tests

6.2.3 Test Requirements

Requirements for the testing are:

1. Internet connection - used for download tested software.
2. Computer with x86 architecture - For testing purpose was chosen machine with x86 architecture.

6.2.4 Results

During testing phase, one significant issue and one smaller issue was discovered. The first issue is caused by the libseccomp library which is not supporting intervals (ranges) in any way. The libseccomp architecture specifies that in one function `seccomp_rule_add` is logical and among partial expressions. Among the functions `seccomp_rule_add` is logical or. Moreover, in one function `seccomp_rule_add` argument position can be mentioned only once which means that it is not possible to write intervals as shown in Figure 6.2.

```c
seccomp_rule_add(
        ctx,
        SCMP_ACT_ALLOW,
        SCMP_SYS(write),
        2,
        SCMP_A0(SCMP_CMP_GE, 1), SCMP_A0(SCMP_CMP_LE, 5) // interval
);
```

Figure 6.2: Possible interval expression

This issue was partially solved in this pull request[11]. The problem with this pull request is that it is not confirmed if it will be merged into libseccomp. Because of that, in this thesis a downstream version of libseccomp which has to be manually installed on the system is used. In the downstream version, interval can be expressed as shown in Figure 6.3.

```c
seccomp_rule_add(
        ctx,
        SCMP_ACT_ALLOW,
        SCMP_SYS(write),
        1,
        SCMP_A0(SCMP_CMP_IN_RANGE, 1, 5) // interval
);
```

Figure 6.3: Proposed operator `SCMP_CMP_IN_RANGE`

Another issue is that the libseccomp operators as `SCMP_CMP_GT/GE/LT/LE` are not designed to work with negative values provided to system calls. This issue is as well reported to the libseccomp project[12]. Luckily this problem is not as critical as the previously stated one. In the end the the kernel is passing the arguments as unsigned and the interpretation of these values is on the implementation of system call.

This situation we can see in particular cases. Very often it is a `mmap` or `llseek`. The problem is nicely visible on the x86_64 / amd64 architecture. The system call numbers are 64bits wide numbers, and BPF used in the kernel uses only 32bits wide registers. Libseccomp is generating a comparison of 64-bit number as a two 32-bits unsigned numbers. We can prove this with libseccomp code or by disassembling BPF instructions generated by libseccomp.

Imagine that we have a clear filter so every system call would be killed. We add only one rule which is shown in Figure 6.4. This rule means that fifth argument must be in the range $[-2, 3]$ including border values.

In Figure 6.5, we can see the generated BPF code. The interesting part begins on line 5. Here we can see that the rule which we specified in C/C++ code begins. On line 6, we store upper 32 bits of a number in accumulator. On line 7 begins we are comparing accumulator with value `0xffffffff` and then on line 8 with zero. This behavior is invalid and should be mentioned in documentation.

[11]https://github.com/seccomp/libseccomp/issues/94
[12]https://github.com/seccomp/libseccomp/issues/69

```
ret |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(mmap), 1,
        SCMP_A4(SCMP_CMP_IN_RANGE, -2, 3)
);
```

Figure 6.4: Only rule we are adding into a libseccomp filter

```
line | CODE | JT | JF | K
=================================
0000: 0x20 0x00 0x00 0x00000004 A = arch
0001: 0x15 0x00 0x0b 0xc000003e if (A != ARCH_X86_64) goto 0013
0002: 0x20 0x00 0x00 0x00000000 A = sys_number
0003: 0x35 0x00 0x01 0x40000000 if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x08 0xffffffff if (A != 0xffffffff) goto 0013
0005: 0x15 0x00 0x07 0x00000009 if (A != mmap) goto 0013
0006: 0x20 0x00 0x00 0x00000034 A = args[4] >> 32 <--- storing upper 32bits
0007: 0x35 0x00 0x05 0xffffffff if (A < 0xffffffff) goto 0013
0008: 0x25 0x04 0x00 0x00000000 if (A > 0x0) goto 0013
0009: 0x20 0x00 0x00 0x00000030 A = args[4] <--------- storing lower 32bits
0010: 0x35 0x00 0x02 0xfffffffe if (A < 0xfffffffe) goto 0013
0011: 0x25 0x01 0x00 0x00000003 if (A > 0x3) goto 0013
0012: 0x06 0x00 0x00 0x7fff0000 return ALLOW
0013: 0x06 0x00 0x00 0x00000000 return KILL
```

Figure 6.5: Disassembled BPF filter on amd64 machine

These two problems are important for the correct output generator of strace2seccomp, and as a result we can not test it properly. As a result of the first issue, there was developed pressure on upstream to merge or implement range macros.

Chapter 7

Conclusion

Bibliography

[1] Fuzzing; brute force vulnerabilty discovery. *Scitech Book News*. vol. 31, no. 4. 2007. ISSN 01966006.

[2] *Clustering*. chapter 12. Wiley-Blackwell. 2009. ISBN 9780470753866. pp. 293–315. doi:10.1002/9780470753866.ch12.
https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470753866.ch12.
Retrieved from:
https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470753866.ch12

[3] *dtrace(1) Linux User's Manual*. November 2017. version *3.1-5.fc26*.

[4] *ftrace(1) Linux User's Manual*. November 2017. version *0.4-56.fc26*.

[5] *ptrace(2) Linux User's Manual*. November 2017. version *4.09*.

[6] *strace(1) Linux User's Manual*. November 2017. version *4.19*.

[7] A. Moss: *Derivatives of parsing expression grammars. Electronic Proceedings in Theoretical Computer Science, EPTCS*. vol. 2. 2017: pp. 180–194. ISSN 20752180. doi:{10.4204/EPTCS.252.18}.

[8] Domingo, D.; Cohen, W.: *SystemTap 3.0*. [Online, accessed 21.2.2018].
Retrieved from:
https://sourceware.org/systemtap/SystemTap_Beginners_Guide.pdf

[9] Drewry, W.: *SECure COMPuting with filters*. [Online, accessed 27.11.2017].
Retrieved from:
https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

[10] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl. *SIGPLAN Not.*. vol. 37, no. 9. September 2002: pp. 36–47. ISSN 0362-1340. doi:10.1145/583852.581483.
Retrieved from: http://doi.acm.org/10.1145/583852.581483

[11] Ganz, J.; Peisert, S.: ASLR. 2017.

[12] Information technology - Programming languages - C++. Standard. International Organization for Standardization. Geneva, CH. December 2017.

[13] Information technology - Syntactic metalanguage - Extended BNF. Standard. International Organization for Standardization. Geneva, CH. March 2011.

[14] Information technology - Programming languages - C. Standard. International Organization for Standardization. Geneva, CH. March 2011.

[15] J. Andrés Díaz-Pace and Marcelo, R, Campo: *ArchMatE: from architectural styles to object-oriented models through exploratory tool support. ACM SIGPLAN Notices.* vol. 40. 2005: page 117. ISSN 03621340. doi:{10.1145/1103845.1094821}.

[16] Lam, D.; Wunsch, D. C.: Chapter 20 - Clustering. In *Academic Press Library in Signal Processing: Volume 1Signal Processing Theory and Machine Learning*, *Academic Press Library in Signal Processing*, vol. 1, edited by R. C. Paulo S.R. Diniz, Johan A.K. Suykens; S. Theodoridis. Elsevier. 2014. pp. 1115 – 1149. doi:https://doi.org/10.1016/B978-0-12-396502-8.00020-6.
Retrieved from:
https://www.sciencedirect.com/science/article/pii/B9780123965028000206

[17] Leventhal, A.; et al.: *About DTrace.* [Online, accessed 2.10.2017].
Retrieved from: http://dtrace.org/blogs/about/

[18] Marhefka, M.: Automatizované fuzz testování aplikací komunikujících přes systém D-Bus. 2013.
Retrieved from: http://hdl.handle.net/11012/55032

[19] markus@chromium.org: *seccompsandbox - overview.wiki.* [Online, accessed 2.10.2017].
Retrieved from:
https://code.google.com/archive/p/seccompsandbox/wikis/overview.wiki

[20] Moore, P.: *Libseccomp.* [Online, accessed 30.11.2017].
Retrieved from: https://github.com/seccomp/libseccomp

[21] Provos, N.: *Systrace - Interactive Policy Generation for System Calls.* [Online, accessed 2.10.2017].
Retrieved from: http://www.citi.umich.edu/u/provos/systrace/

[22] Schmidt, R. F.: Chapter 20 - Software Acceptance Testing. In *Software Engineering*, edited by R. F. Schmidt. Boston: Morgan Kaufmann. 2013. ISBN 978-0-12-407768-3. pp. 335 – 341. doi:https://doi.org/10.1016/B978-0-12-407768-3.00020-3.
Retrieved from:
https://www.sciencedirect.com/science/article/pii/B9780124077683000203

[23] Schubert, E.; Sander, J.; Ester, M.; et al.: DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.*. vol. 42, no. 3. July 2017: pp. 19:1–19:21. ISSN 0362-5915. doi:10.1145/3068335.
Retrieved from: http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/3068335

[24] Schulist, J.; Borkmann, D.; Starovoitov, A.: *Linux Socket Filtering aka Berkeley Packet Filter (BPF).* [Online, accessed 11.12.2017].
Retrieved from:
https://www.kernel.org/doc/Documentation/networking/filter.txt

[25] Silberschatz, A.; Galvin, P. B.; Gange, G.: *Operating System Concepts.* Hoboken, NJ: Wiley. 9 edition. 2013. ISBN 9781118063330.

[26] Sroka, R.: *Extend USBGuard to Support External Authorization Policy Sources*. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. 2018.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=21006

[27] Takanen, A.: Fuzzing: the Past, the Present and the Future.

[28] Takanen, A.; DeMott, J.; Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc.. first edition. 2008. ISBN 1596932147, 9781596932142.

[29] Team, P. E. P.: PaX Address Space Layout Randomization (ASLR). 2001. [Online, accessed 21.4.2018].
Retrieved from: https://pax.grsecurity.net/docs/aslr.txt

[30] Vimpari, M.: An evaluation of free fuzzing tools. 2015.
Retrieved from: http://jultika.oulu.fi/files/nbnfioulu-201505211594.pdf

Appendix A

Comparison of libseccomp and raw BPF filtering

A.1 BPF

```
int myapp_seccomp_raw_start(void)
{
struct sock_filter filter[] = {
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 4),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 0x00, 0x12),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 0),
       BPF_STMT(BPF_JMP+BPF_JGE+BPF_K, 0x40000000, 0x10, 0x00),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_open , 0x0e, 0x00),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_close, 0x0d, 0x00),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_read, 0x00, 0x0d),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 20),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x0b),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 16),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x09),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 28),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x02),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 24),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x05, 0x00),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 36),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, (SSIZE_MAX >> 32), 0x00, 0x02),
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 32),
       BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, (SSIZE_MAX & 0xffffffff), 0x01, 0x00),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
};
struct sock_fprog prog = {
       .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
       .filter = filter,
};
if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) < 0)
```

```
        return -errno;
if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) < 0)
        return -errno;
return 0;
}
```

Listing A.1: Using raw BPF filtering

A.2 libseccomp

```
int myapp_libseccomp_start(void)
{
        int rc;
        scmp_filter_ctx ctx;
        ctx = seccomp_init(SCMP_ACT_KILL);

        if (ctx == NULL)
                return -ENOMEM;

        rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(open), 0);

        if (rc < 0)
                goto out;

        rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);

        if (rc < 0)
                goto out;

        rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 3,
                SCMP_A0(SCMP_CMP_EQ, STDIN_FILENO),
                SCMP_A1(SCMP_CMP_NE, 0x0),
                SCMP_A2(SCMP_CMP_LT, SSIZE_MAX)
        );

        if (rc < 0)
                goto out;

        rc = seccomp_load(ctx);

out:
        seccomp_release(ctx);
        return rc;
}
```

Listing A.2: Using simpler libseccomp wrapper

Appendix B

Output of strace2seccomp

B.1 Example Output no.1

```
/*
 * Generated seccomp template with initialized filter using st2se.
 * link with -lseccomp
 */

#ifdef __cplusplus
   #define NULL nullptr
   #include <cstdio>
#else
   #include <stdio.h>
#endif

#include <seccomp.h>

int setup_seccomp_whitelist(){

   scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);
   int rc = 0;

   // seccomp rules
   //-----------------------------------------------------------------------
   rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(access), 2,
       SCMP_A0(SCMP_CMP_EQ, 0),
       SCMP_A1(SCMP_CMP_IN_RANGE, 0, R_OK)
   );
   rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(arch_prctl), 1,
       SCMP_A0(SCMP_CMP_IN_RANGE, 0, ARCH_SET_FS)
   );
   rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);
   rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(execve), 2,
       SCMP_A0(SCMP_CMP_EQ, 0),
       SCMP_A1(SCMP_CMP_EQ, 0)
```

```
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 1,
        SCMP_A0(SCMP_CMP_GE, 0), SCMP_A0(SCMP_CMP_LE, 0)
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(mmap), 5,
        SCMP_A1(SCMP_CMP_IN_RANGE, 0, 14576),
        SCMP_A2(SCMP_CMP_IN_RANGE, 0, PROT_READ|PROT_WRITE),
        SCMP_A3(SCMP_CMP_IN_RANGE, 0, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS),
        SCMP_A4(SCMP_CMP_IN_RANGE, -1, 0),
        SCMP_A5(SCMP_CMP_IN_RANGE, 0, 0)
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(mprotect), 2
        SCMP_A1(SCMP_CMP_IN_RANGE, 0, 4096),
        SCMP_A2(SCMP_CMP_IN_RANGE, 0, PROT_READ)
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(mprotect), 2
        SCMP_A1(SCMP_CMP_IN_RANGE, 16384, 2093056),
        SCMP_A2(SCMP_CMP_IN_RANGE, 0, PROT_READ)
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(munmap), 1
        SCMP_A1(SCMP_CMP_IN_RANGE, 0, 213488)
    );
    rc |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(openat), 3
        SCMP_A0(SCMP_CMP_IN_RANGE, 0, AT_FDCWD),
        SCMP_A1(SCMP_CMP_EQ, 0),
        SCMP_A2(SCMP_CMP_IN_RANGE, 0, O_RDONLY|O_CLOEXEC)
    );
    //------------------------------------------------------------------------

    if (rc != 0) {
        goto out;
    }

    if (seccomp_load(ctx) != 0) {
        rc = 2, goto out;
    }
    return rc;
out:
    seccomp_release(ctx);
    return rc;
}

int main()
{
    // setup and load seccomp whitelist
    if (setup_seccomp_whitelist() != 0) {
        return 1;
    }
    // Put your code below
    return 0;
}
```
Listing B.1: Example output of strace2seccomp