Automating Seccomp Filter Generation for Linux Applications

Claudio Canella\textsuperscript{1}, Mario Werner\textsuperscript{1}, Daniel Gruss\textsuperscript{1}, Michael Schwarz\textsuperscript{2}

\textsuperscript{1}Graz University of Technology \textsuperscript{2}CISPA Helmholtz Center for Information Security
Motivation

- Memory safety vulnerabilities are common
Motivation

- Memory safety vulnerabilities are common
- Sandboxing helps in limiting their impact
• Memory safety vulnerabilities are common
• Sandboxing helps in limiting their impact
• Linux seccomp: works but hard to do
Motivation

- Memory safety vulnerabilities are common
- Sandboxing helps in limiting their impact
- Linux seccomp: works but hard to do

→ Can we automate seccomp sandboxing?
Linux Seccomp

App.

Kernel

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Linux Seccomp

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
```c
int main(int argc, char* argv[]) {
    int infd, outfd;
    ssize_t read_bytes;
    char buffer[1024];

    printf("Copying '%s' to '%s'\n", argv[1], argv[2]);
    if((infd = open(argv[1], O_RDONLY)) > 0) {
        if((outfd = open(argv[2], O_WRONLY | O_CREAT, 0644)) > 0) {
            while((read_bytes = read(infd, &buffer, 1024)) > 0)
                write(outfd, &buffer, (ssize_t)read_bytes);
        }
    }
    close(infd);
    close(outfd);
    return 0;
}
```
Challenges

 Syscalls from C Functions?
Challenges

Syscalls from C Functions?  Entire Code Base?
Challenges

- Syscalls from C Functions?
- Entire Code Base?
- Third-party Libraries?
P1: Static Analysis
P1: Static Analysis

Source

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
P1: Static Analysis

Source Analyzer

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
P1: Static Analysis

Source Analyzer

Annotated Binary File(s)
P1: Static Analysis

Source Analyzer

Annotated Binary File(s)
P1: Static Analysis

Source Analyzer

Annotated Binary File(s)

Binary Analyzer

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
P1: Static Analysis

<>
Source Analyzer

Binary Analyzer

Annotated Binary File(s)

P2: Dynamic Refinement

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
P1: Static Analysis

- Source Analyzer
- Binary Analyzer

Annotated Binary File(s)

P2: Dynamic Refinement

- Dynamic Analyzer

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
P1: Static Analysis

Source Analyzer

Annotated Binary File(s)

P2: Dynamic Refinement

Dynamic Analyzer

Annotated Binary File(s)

Chestnut Patcher

Sandboxed Binary
P1: Static Analysis

Source Analyzer

Annotated Binary File(s)

P2: Dynamic Refinement

Dynamic Analyzer

Annotated Binary File(s)

Chestnut Patcher

Sandboxed Binary

or

Chestnut Generator

Wrapped Binary

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Sources
Sourcalyzer

Sources → Extract System Call Number → Build Call Graph → Annotated Binary File(s)

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Sourcalyzer

- Sources
- Extract System Call Number
- Build Call Graph
- Libraries
- Generate System Call List
- Annotated Binary File(s)

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Sourcalyzer

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
```c
#include <stdio.h>

int main() {
    puts("Hello World!");
}
```

```
puts(const char *s) {
    int r; flock(stdout);
    r = -(fputs(s, stdout) < 0 || putc_unlocked(\n', stdout) < 0);
    funlock(stdout); return r;
}
```
#include <stdio.h>

int main() {
    puts("Hello World!");
}

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
```c
#include <stdio.h>
int main() {
    puts("Hello World!");
}
```
# include <stdio.h>
int main() {
    puts("Hello World!");
}

// musl/src/stdio/puts.c
int puts(const char *s) {
    int r; FLOCK(stdout);
    r = -(fputs(s, stdout) < 0 ||
          putc_unlocked(\n, stdout) < 0);
    FUNLOCK(stdout); return r;
}
```c
#include <stdio.h>
int main() {
    puts("Hello World!");
}
```
Call Graph Example

```c
#include <stdio.h>
int main () {
    puts("Hello World!");
}

// musl/src/stdio/puts.c
int puts(const char *s) {
    int r; FLOCK(stdout);
    r = -(fputs(s, stdout) < 0 || putc_unlocked(\n', stdout) < 0);
    FUNLOCK(stdout); return r;
}

{"call_targets": ["__lockfile",
    "fputs",
    "__overflow",
    "__unlockfile"],

"name": "puts",
"type": "i32 (i8*)"}
```

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Call Graph Example

main → puts

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Call Graph Example

- main
- puts
- __lockfile
- __unlockfile
- fputs
- __overflow

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Call Graph Example

main → puts

__lockfile  __unlockfile  fputs  __overflow

fwrite  strlen

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Call Graph Example

main

.puts

.lockfile

.unlockfile

.fputs

.overflow

.memcpy

 fwrite

.strlen

fwrite

strlen

memcpy

fputs

overflow

unlockfile

lockfile

puts

main

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Call Graph Example

- main
- puts
  - __lockfile
  - __unlockfile
  - fwrite
  - memcpy
- fputs
- __overflow
- fwrite
- strlen
  - i64 (FILE*, i8*, i64)
  - __stdout_write
    - 16
  - __stdio_write
    - 20

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
• Extract system calls from existing binaries/libraries
• Extract system calls from existing binaries/libraries
• Capstone: disassemble binary
Binalyzer: Overview

- Extract system calls from **existing** binaries/libraries
- Capstone: **disassemble** binary
- Anger: build call graph
```assembly
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall
```
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall
rax = ?
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall
rax = ?
rax = ?
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall
rax = ?
rax = ?
rax = ?
mov  $0x1,%bl
xor  %edi,%edi
mov  %ebx,%eax
lea  0xf(%rip),%rsi
mov  $0xd,%edx
syscall

rax = ?
rax = ?
rax = ?
```assembly
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall
rax = rbx = ?
```

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xd,%edx
syscall

rax = rbx = ?
rax = rbx = ?
rax = ?
rax = ?
rax = ?
mov $0x1, %bl
xor %edi, %edi
mov %ebx, %eax
lea 0xf(%rip), %rsi
mov $0xd, %edx
syscall
rax = rbx = ?
rax = rbx = ?
rax = ?
rax = ?
rax = ?
mov $0x1,%bl
xor %edi,%edi
mov %ebx,%eax
lea 0xf(%rip),%rsi
mov $0xad,%edx
syscall

rax = rbx = $0x1
rax = rbx = ?
rax = rbx = ?
rax = ?
rax = ?
System Call Mapping

puts

fork

sys_writev
sys_clone
sys_futex
sys_wait4
sys_execve
sys_rt_sigprocmask

system
Finalyzer: Overview

- **Strace-like system**
- **Strace-like** system
- Dynamically trace system calls
Finalyzer: Overview

- **Strace-like system**
- Dynamically trace system calls
- Automatically add *missed system calls* or optionally remove *never-used* ones
Kernel

Tracee

notify
allow
Log
Syscall
System Call
return
Finalyzer

```
www.tugraz.at
```

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
• Performance, Functional Correctness, and Security
Evaluation

- Performance, Functional Correctness, and Security
- Analyzed Client, Server, and Database applications
Evaluation

- Performance, Functional Correctness, and Security
- Analyzed Client, Server, and Database applications
- 18 applications
Evaluation: Performance

- Worst Compile Time Overhead (git): 28% (+19s)
Evaluation: Performance

- Worst Compile Time Overhead (git): 28% (+19 s)
- Worst Binary Extraction Time (ffmpeg): 11 min
Evaluation: Seccomp

- Seccomp has inherent performance impact $\rightarrow$ nothing Chestnut can do about that
- Recent work (Linux 5.11) improved performance
Evaluation: Functional Correctness

- Use application testsuites for checks
Evaluation: Functional Correctness

- Use application testsuites for checks
- Code coverage metrics for better estimations of correctness
Evaluation: Functional Correctness

- Use application testsuites for checks
- Code coverage metrics for better estimations of correctness
  - Line coverage: 59-77%
  - Function coverage: 61-92%
Evaluation: Functional Correctness

- Use application testsuites for checks
- Code coverage metrics for better estimations of correctness
  - Line coverage: 59-77 %
  - Function coverage: 61-92 %
- Observed no crashes in tests
Evaluation: Functional Correctness

- Use application **testsuites** for checks
- **Code coverage metrics** for better estimations of correctness
  - Line coverage: 59-77%
  - Function coverage: 61-92%
- Observed **no crashes** in tests
- 6 month long-term study using nginx: **no crashes**
• Avg. Number of blocked system calls:
Evaluation: Security

- Avg. Number of blocked system calls:
  - exec: 9 (50%)
  - mprotect: 0 (0%)
  - 14 (78%)
  - 11 (61%)
  - 302 (87%)
  - 288 (83%)

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz
Evaluation: Security

- Avg. Number of blocked system calls:
  - `read`: 302 (87%)
  - `write`: 288 (83%)
- `exec` system calls blocked:
• Avg. Number of blocked system calls:
  - : 302 (87 %)
  - : 288 (83 %)

• exec system calls blocked:
  - : 9 (50 %)
  - : 14 (78 %)
Evaluation: Security

- Avg. Number of blocked system calls:
  - : 302 (87%)
  - : 288 (83%)

- `exec` system calls blocked:
  - : 9 (50%)
  - : 14 (78%)

- `mprotect` system calls blocked:
Evaluation: Security

- Avg. Number of blocked system calls:
  - `<file>`: 302 (87%)
  - `<gear>`: 288 (83%)
- `exec` system calls blocked:
  - `<file>`: 9 (50%)
  - `<gear>`: 14 (78%)
- `mprotect` system calls blocked:
  - `<file>`: 11 (61%)
  - `<gear>`: 0 (0%)
• 175 CVEs extracted from mitre database
• 175 CVEs extracted from mitre database
• Full CVEs:
• 175 CVEs extracted from mitre database
• Full CVEs:
  • : 64 %
  • : 62 %
• 175 CVEs extracted from mitre database
• Full CVEs:
  </> : 64 %
  ⌨️ : 62 %
• Subvariants:
  </> : 75 %
  ⌨️ : 72 %
You can find our proof-of-concept implementation of Chestnut on:

- https://github.com/IAIK/Chestnut
More details in the paper

- More detailed security evaluation
- Information on overapproximation
- More implementation details
- ...

**CCSW [Can+21]**

Claudio Canella, Mario Werner, Daniel Gruss, Michael Schwarz.
Automating Seccomp Filter Generation for Linux Applications.
• Reduced time-consuming, manual analysis to automated process
Recap

• Reduced time-consuming, manual analysis to **automated process**
• Showed that we can **improve overall system security**
• Reduced time-consuming, manual analysis to **automated process**
• Showed that we can **improve overall system security**
• Chestnut only has **small performance impact**
Automating Seccomp Filter Generation for Linux Applications

Claudio Canella¹, Mario Werner¹, Daniel Gruss¹, Michael Schwarz²

¹Graz University of Technology ²CISPA Helmholtz Center for Information Security
References

This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM, Intel, and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.