Finding Injection Vulnerabilities

ERICSSON

Improvements of the Taint Analysis of the Clang Static Analyzer

Dániel Krupp
daniel.krupp@ericsson.com
Ericsson

2026-04-10

Real issue found in curl (test code)

```
187 {
188   FILE *fp = fopen(configfile, FOPEN_READTEXT);
          3  Taint originated here
          4  Taint propagated to the return value
          5  Assuming that 'fopen' is successful
189   resetdefaults();
190   if(fp) {
191     char buffer[512];
192     logmsg("parse config file");
193     while(fgets(buffer, sizeof(buffer), fp)) {
            6  Taint propagated to the 1st argument
            7  Entering loop body
194       char key[32];
195       char value[260];
196       if(2 == sscanf(buffer, "%31s %259s", key, value)) {
              8  Taint propagated to the 3rd argument, 4th argument
              9  Assuming the condition is true
197         if(!strcmp(key, "version")) {
              10  Taint propagated to the return value
              11  Assuming the condition is false
198           config.version = byteval(value);
199           logmsg("version [%d] set", config.version);
200         }
201         else if(!strcmp(key, "nmethods_min")) {
              12  Taint propagated to the return value
              13  Assuming the condition is false
202           config.nmethods_min = byteval(value);
203           logmsg("nmethods_min [%d] set", config.nmethods_min);
204         }
205         else if(!strcmp(key, "nmethods_max")) {
              14  Taint propagated to the return value
              15  Assuming the condition is false
206           config.nmethods_max = byteval(value);
207           logmsg("nmethods_max [%d] set", config.nmethods_max);
208         }
209         else if(!strcmp(key, "backend")) {
              16  Taint propagated to the return value
              17  Assuming the condition is true
210           strcpy(config.addr, value);
                18  Unrestricted copy of untrusted data can cause buffer overflow(CERT/STR31-C. Sanitize the data or use of strncpy/strncat/...)
                For more information see the checker documentation.
```

```
struct configurable {
  unsigned char version; /* initial version byte in the request must match
                            this */
  unsigned char nmethods_min; /* minimum number of nmethods to expect */
  unsigned char nmethods_max; /* maximum number of nmethods to expect */
  unsigned char responseversion;
  unsigned char responsemethod;
  unsigned char reqcmd;
  unsigned char connectrep;
  unsigned short port; /* backend port */
  char addr[32]; /* backend IPv4 numerical */
  char user[256];
  char password[256];
};
```

configurable.addr is 32 bytes
The attacker can write in up to 259 bytes!

Clang Static Analyzer report displayed in CodeChecker

Common Weakness Enumerations involving injection vulnerabilities

ERICSSON

- CWE-78 OS Command Injection
- CWE-134 Format String Vulnerability
- CWE-90 LDAP Injection
- CWE-606 Unchecked Input for Loop Condition
- CWE-89 SQL Injection
- CWE-121 / 122 Stack/Heap-based Buffer Overflow
- …

What is available today?

ERICSSON

Clang Static Analyzer has an optin.taint checker package

- optin.taint
  - optin.taint.GenericTaint (C, C++)
    - Finds generic taint source -> sink data flows.
    - YAML Configuration file: taint sources, sinks, propagation rules
  - optin.taint.TaintedAlloc (C, C++)
    - Finds malloc(..) calls with tainted and unbounded memory size
  - optin.taint.TaintedDiv (C, C++, ObjC)
    - Finds division by tainted values
- **Why it was alpha?**
  - The checker diagnostics were missing the full path from the source to the sink.
  - False positives on some tainted integer values
  - https://github.com/llvm/llvm-project/pull/67352 (2024)

A different reporting philosophy

ERICSSON

```
int avg(int *numbers, int len){
  int sum = 0;
  for (int i=0; i < len; i++){
    sum+=numbers[i];
  }
  return sum/len;
}
```

Should we report a division by zero?

No, as we don't see a clue that `len` can be 0!

In general, we only report a problem, if we can prove that on the specific execution path there _is_ an error.

A different reporting philosophy

ERICSSON

```
int avg(int *numbers, int len){
  int sum = 0;
  for (int i=0; i < len; i++){// Loop Condition is a tainted, attacker-controlled value
    sum+=numbers[i];
  }
  return sum/len; // Division by a tainted, possible zero value
}
int vulnerable(void){
  int len;
  int numbers[]={1,2,3,4,5};
  scanf("%d", &len);
  return (avg(numbers, len));
}
```

When we see that a value is tainted and we cannot prove that its usage is safe, we must warn!

Note: we cannot prove either that the operations result in an error

Juliet Test Suite

ERICSSON

- Juliet C/C++ 1.3
- A collection of test cases in the C/C++ language. It contains examples organized under 118 different CWEs.
- The C/C++ part contains 64 099 test cases and more than 100 000 files.
- https://samate.nist.gov/SARD/test-suites/112

- **~1000 test cases per CWE**
- **Multiple functional variants per TC (different type of sources and sinks)**
- **Gradually more difficult control/data flow variants**
- **Tests for false positives**
- **Multi-Translation unit tests!**

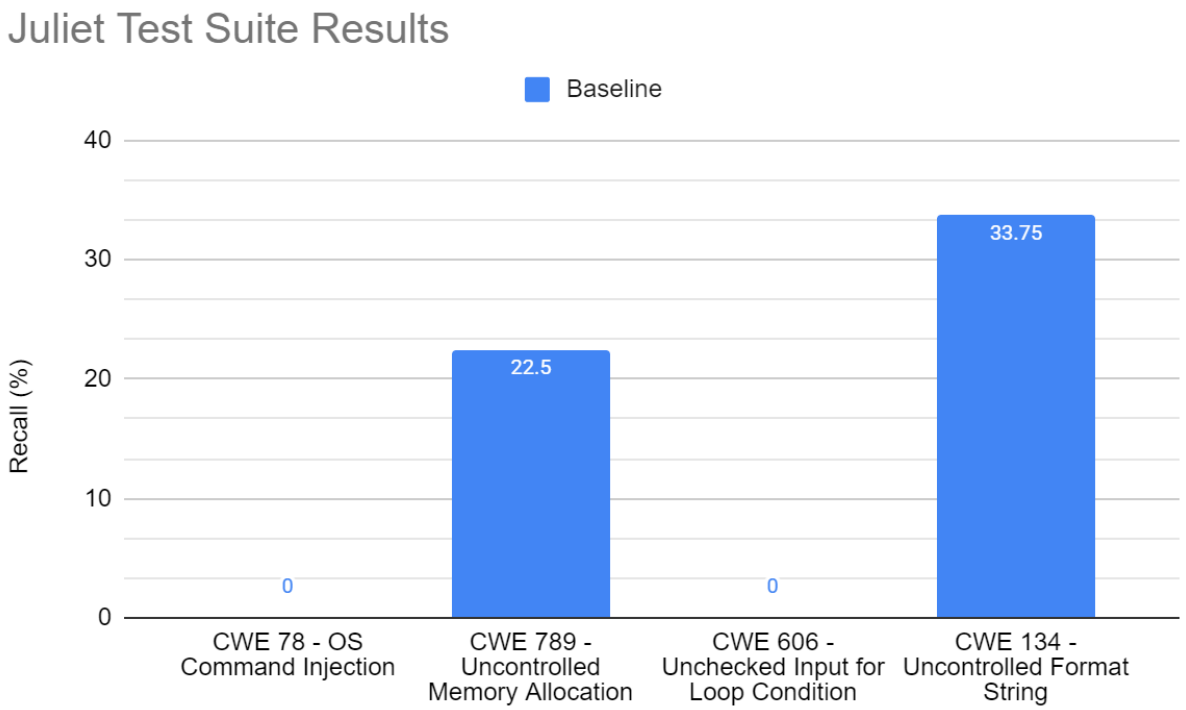
We executed the tests on Linux only

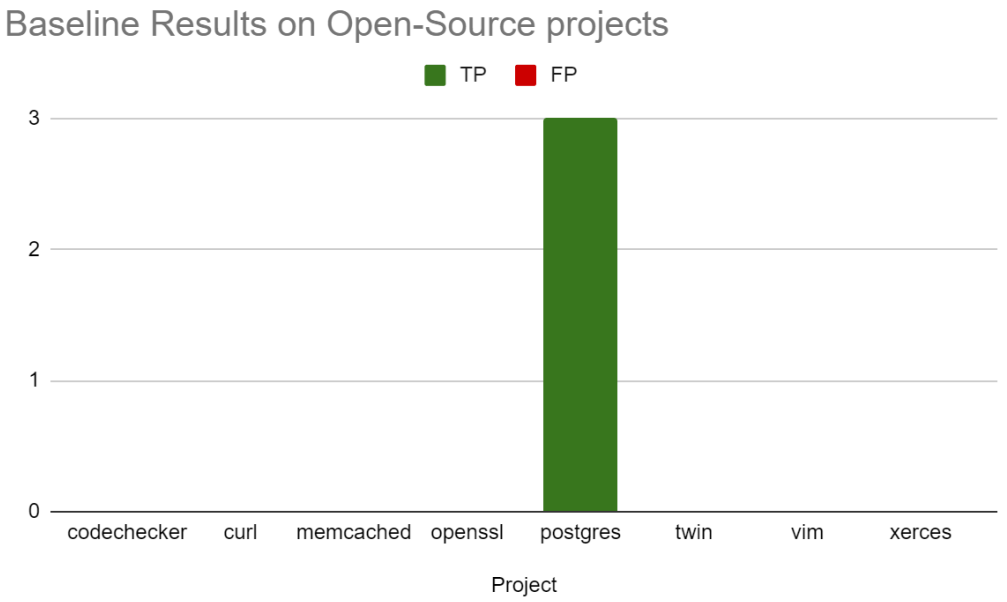
- Windows and wchar test cases were excluded (but the analyzer supports them)

NIST
National Institute of
Standards and Technology

Where are we now?

- Small number of reports in Open-Source projects
  - Not too many false positives though…
- Small baseline coverage on the Juliet Test suite





Improvement opportunities

ERICSSON

Some sources and sinks are missing

- Partially tainted buffers passed to sinks are not reported
- Dangerous string handling functions (strcpy, strcat, etc.) are not sinks
- Tainted Loops not detected (CWE-606**)**
- Incoming parameters of the main function not handled as taint sources

Taint Propagation problems

- Taintedness property is getting lost during propagation
- C++ virtual function calls not followed through TU boundaries

No infrastructure to perform taint analysis of library interfaces

Partially tainted buffers passed to sinks are not reported – CWE 78

```
char data_buf[100] = "ls ";
char *data = data_buf;
int recvResult;
struct sockaddr_in service;
SOCKET connectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&service, 0, sizeof(service));
service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr(IP_ADDRESS);
service.sin_port = htons(TCP_PORT);
connect(connectSocket, (struct sockaddr*)&service, sizeof(service));
size_t dataLen = strlen(data);
recvResult = recv(connectSocket, (char *)(data + dataLen), sizeof(char) * (100 - dataLen - 1), 0);
system(data);// expected-warning {{Untrusted data is passed to a system call}}
```

Extension of the sink handling.

https://github.com/dkrupp/llvm-project/tree/taint_partial_array

Without +dataLen we would have found it.

Dangerous string handling functions (strcpy, strncpy, etc.) are not sinks

```
//strcpy: src should be a taint sink
void test_strcpy() {
  char filename[1024];
  char txt[2048];
  scanf("%s", txt);
  clang_analyzer_isTainted(*txt); // expected-warning{{YES}}
  strcpy(filename, txt);// expected-warning {{Unrestricted copy of untrusted
data can cause buffer overflow}}
}
```

If the buffer size of the src is tainted, report it!

https://github.com/dkrupp/llvm-project/tree/tainted_stringfuns

Dangerous string handling functions (strcpy, strncpy, etc.) are not sinks

ERICSSON

- int **snprintf**(char * s, **size t n**, const char * format, ...)
- int **vsprintf**(char * s, const char * format, **va list arg**)
- int **vsnprintf**(char * s, **size t n**, const char * format, va list arg)
- char * **strcpy**(char * destination, **const char * source**)
- char * **strncpy**(char * destination, const char * source, **size t num**)
- size t **strlcpy**(char dst, const char *src, **size t size**)
- char * **strcat**(char * destination, **const char * source**);
- char * **strncat**(char * destination, const char * source, **size t num**)
- size t **strlcat**(char dst, const char *src**, size t size**)

The bold parameters are added as taint sink to the GenericTaint checker currently.

Reporting potential buffer overflows

ERICSSON

```
int main(int argc, char *argv[]){
   if (argc < 2)
     return 1;
   char buf[2048];
   strcpy(buf, argv[1]); // Potential buffer overflow!
}
```

```
int main(int argc, char *argv[]){
   if (argc < 2)
     return 1;
   char* buf=(char*) malloc(strlen(argv[1]) + 1);
   strcpy(buf, argv[1]); // Don't report this !
}
```

Differentiation would be needed to decrease false positive findings (work in progress).

Tainted Loops not detected (CWE-606)

ERICSSON

```
void test_tainted_for(void){
  long limit;
  scanf("%ld", &limit);
  //if (limit > 1000)
  //  return;
  for (int i=0; i < limit; i++){  // Loop condition is a tainted, attacker
                                  //   controlled value
    printf("%d ",i);
  }
```

New optin.taint.TaintedLoop checker.
https://github.com/dkrupp/llvm-project/tree/tainted_loop

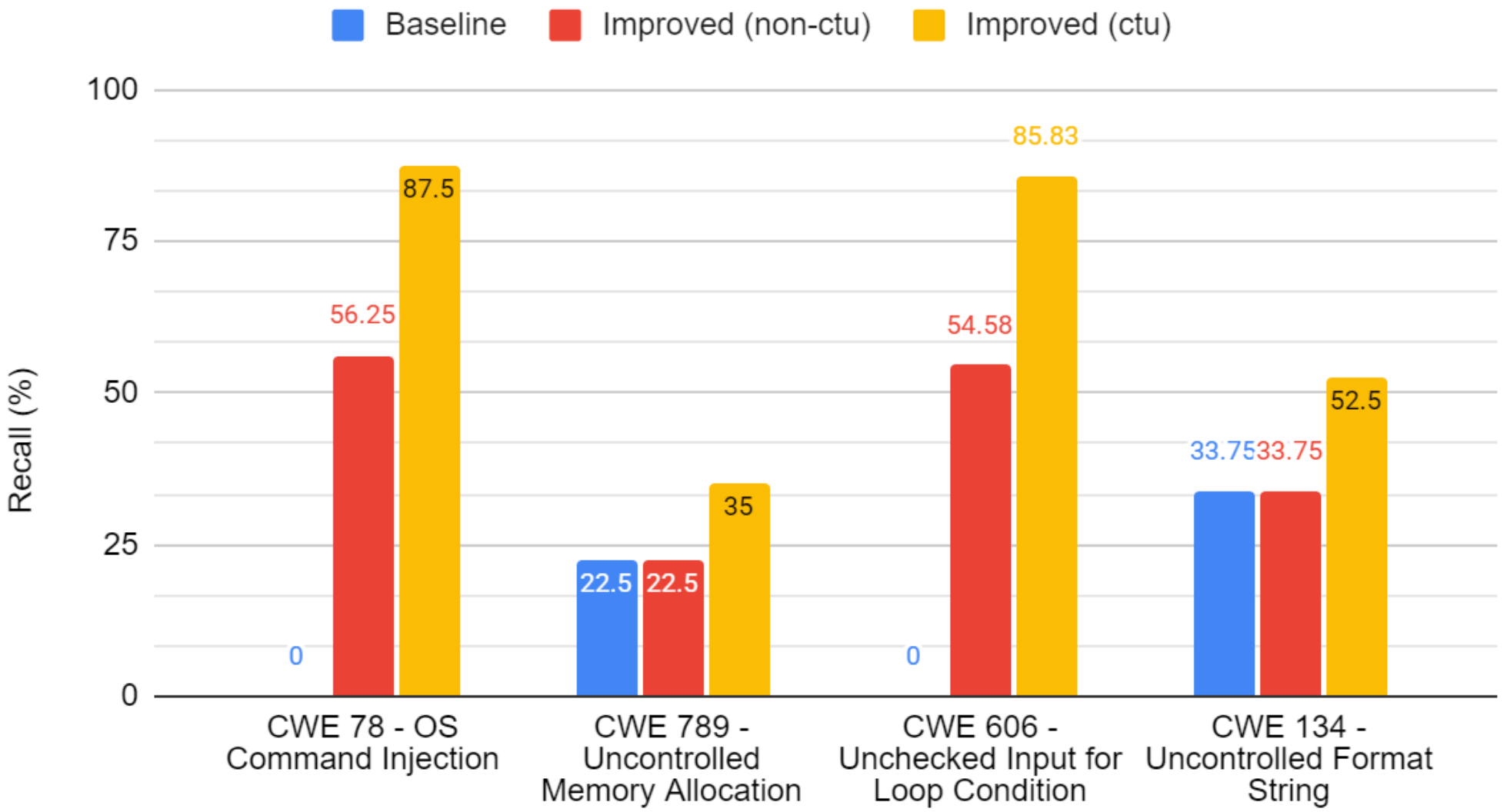
Incoming parameters of the main function not handled as taint sources

```
int main(int argc, char *argv[]){
   if (argc < 2)
     return 1;
   char cmd[2048] = "/bin/cat ";
   strncat(cmd, argv[1], sizeof(cmd) - strlen(cmd)- 1);
   system(cmd);// Untrusted data is passed to a system call
   return 0;
}
```

The incoming parameters of the program should be considered tainted
If the analyzer option **assume-controlled-environment** is set to false

PR already accepted. https://github.com/llvm/llvm-project/pull/178054

Juliet Test Suite Results



CTU - Cross Translation Unit Analysis

With 100% Precision

Taint propagation

ERICSSON

Tainted value is getting propagated as the flow of values are tracked by the analyzer

If a function is called, which cannot be analyzed (inlined) the assumptions on writable parameters, global values get lost

- Too complex function
- Definition of the function is not known
- The function was inlined already too many times
- Too deep stack
- Analysis budget is out
- …

New propagation modes

ERICSSON

When the analyzer encounters a function that is not inlined, not modeled by the analyzer and not described by the user in the taint configuration…

- **Forget mode** - **Current baseline mode**. Taintedness gets lost if the called function was analyzed conservatively.
- **Keep mode –** If a writeable parameter was tainted, it's taintedness is kept, but not spread to other writeable parameters or to the return value.
- **Spread mode –** If a parameter is tainted, the taintedness would spread to all other parameters and to the return value.

There are many functions like this!

Taintedness gets lost at all unmodeled function calls in forget mode

ERICSSON

```
void test_lost_printf(){
  char cmd[2048] = "/bin/cat ";
  char *filenameOnHeap = (char*) malloc(1024);
  scanf("%s", filenameOnHeap);
  printf("tainted input:%s\n",filenameOnHeap); // taintedness gets lost here
  clang_analyzer_isTainted(*filenameOnHeap); // forget-warning{{NO}}
  strncat(cmd, filenameOnHeap, sizeof(cmd) - 1);
  clang_analyzer_isTainted(*cmd); // forget-warning{{NO}}
  system(cmd);// Warning lost
  free(filenameOnHeap);
}
```

Taintedness preserved in keep/spread mode

ERICSSON

```
void test_lost_printf(){
  char cmd[2048] = "/bin/cat ";
  char* filenameOnHeap = (char*) malloc(1024);
  scanf("%s", filenameOnHeap);
  printf("tainted input:%s\n",filenameOnHeap);
  clang_analyzer_isTainted(*filenameOnHeap); // forget-warning{{NO}}
                                             // keep-warning@-1{{YES}}
                                             // spread-warning@-2{{YES}}
  strncat(cmd, filenameOnHeap, sizeof(cmd) - 1);
  clang_analyzer_isTainted(*cmd); // forget-warning{{NO}}
                                  // keep-warning@-1{{YES}}
                                  // spread-warning@-2{{YES}}
  system(cmd);// keep-warning {{Untrusted data is passed to a system call}}
              // spread-warning@-1 {{Untrusted data is passed to a system call}}
  free(filenameOnHeap);
}
```

What can we do (in forget mode)?

ERICSSON

- **Add a propagation configuration rule** for each such conservatively evaluated function
- Variadic functions such as printf cannot be expressed
- How would I know which function was evaluated conservatively?
- **How many potential warnings do get lost in a real project?**

```
Propagations:
 # Propagation function
  #   char *dirname(char *path)
  #
  # Result example:
  #   char* path = read_path();
  #   char* dir = dirname(path);
  #   // dir is tainted if path was tainted
  - Name: dirname
    SrcArgs: [0]
    DstArgs: [-1]
```

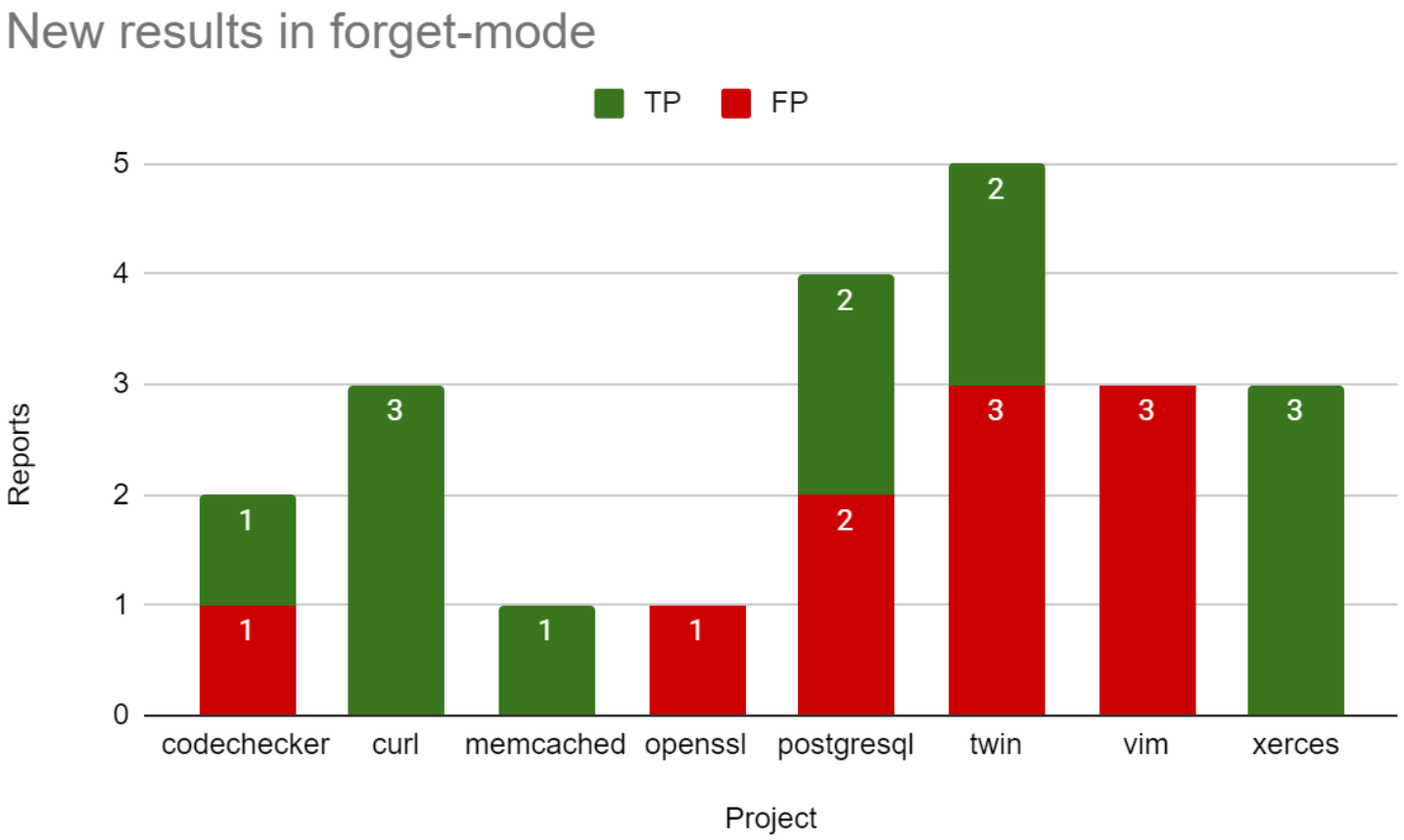
Spread mode uncovers potential vulnerabilities (with more false positives)

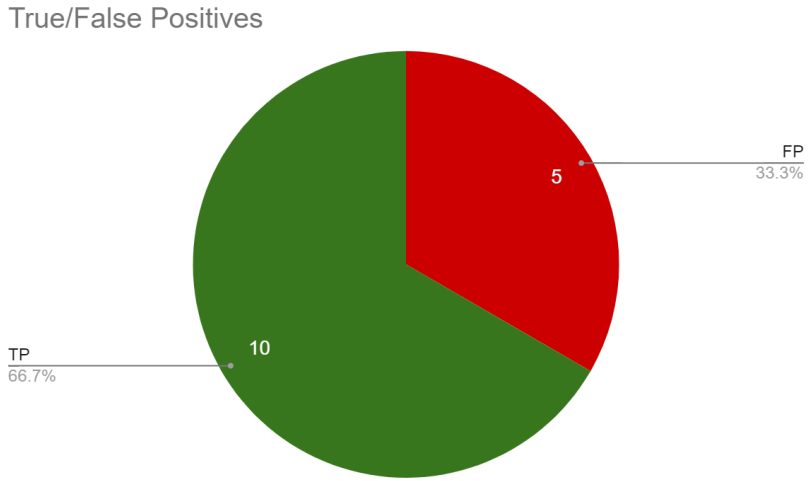
```
void unknownTransformInOut(char *in, char *out);
void test_unknown_transform2(int input){
  char cmd[2048] = "/bin/cat ";
  char filename[1024];
  char filename_transformed[1024];
  scanf ("%s",filename);
  unknownTransformInOut(filename, filename_transformed); // spread propagation
  clang_analyzer_isTainted(*filename); // expected-warning{{YES}}
  clang_analyzer_isTainted(*filename_transformed); // spread-warning{{YES}}
                                                   // keep-warning@-1{{NO}}
  strncat(cmd, filename_transformed, sizeof(cmd) - 1);
  system(cmd);// spread-warning {{Untrusted data is passed to a system call}}
}
```

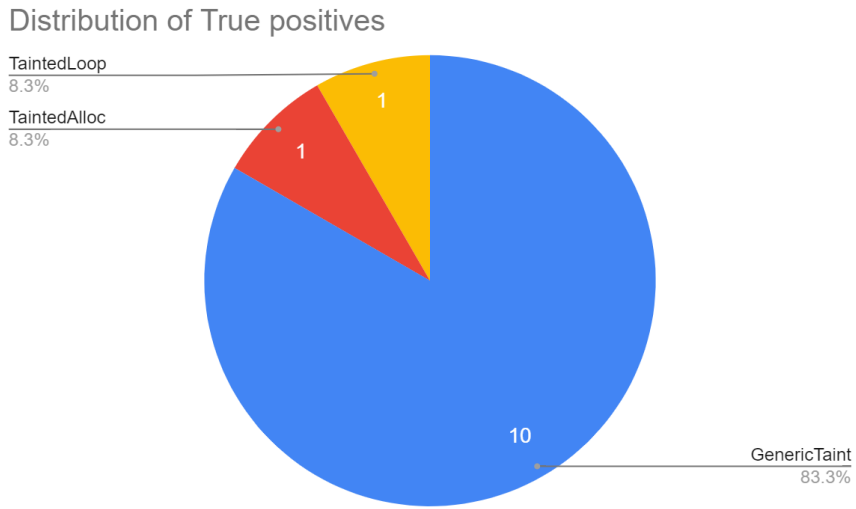
Spreads taintedness aggressively, but only if the called function cannot be inlined.

Extra false positives on impossible data flow paths. Still could be worth it for a security analyst.

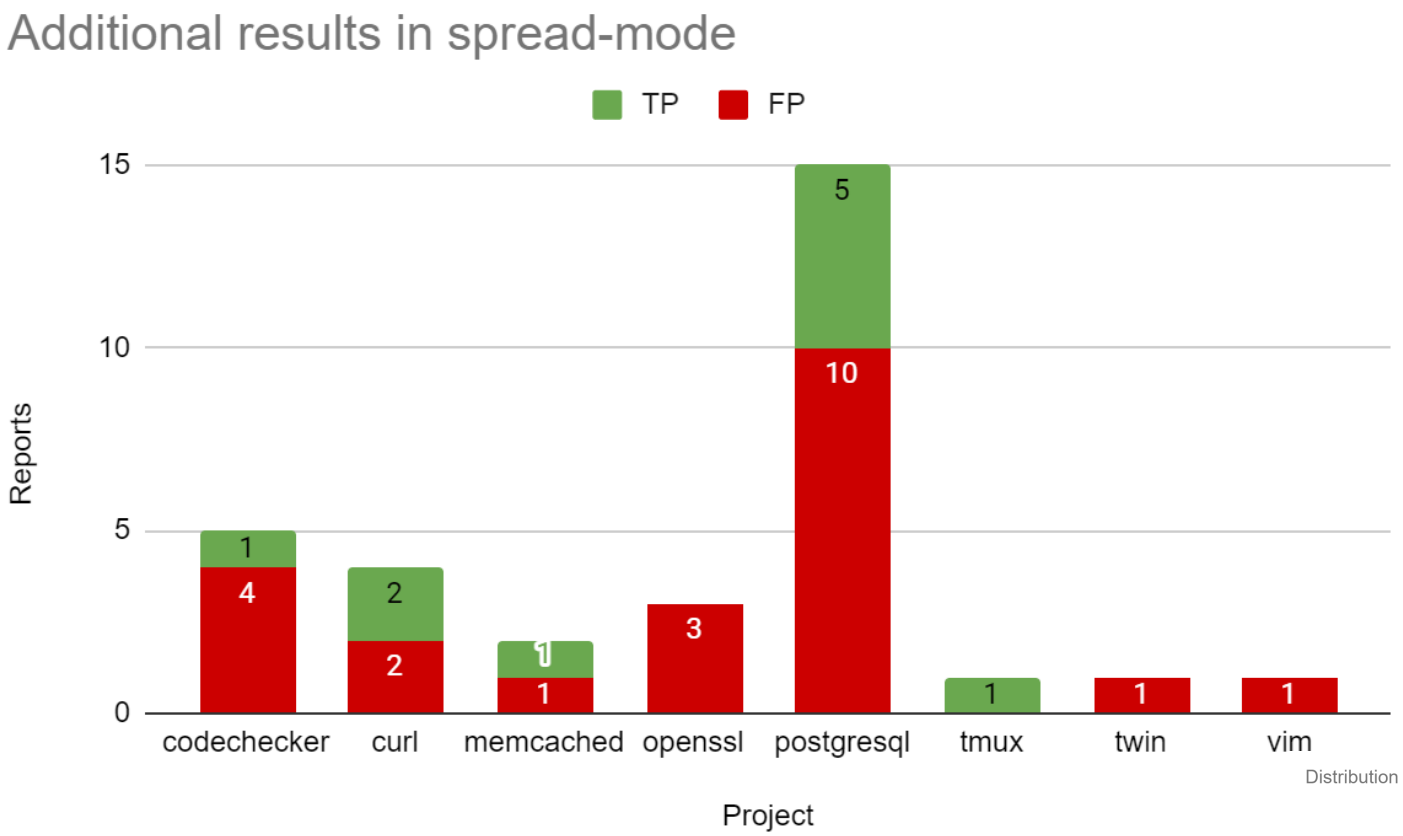
New results in forget mode

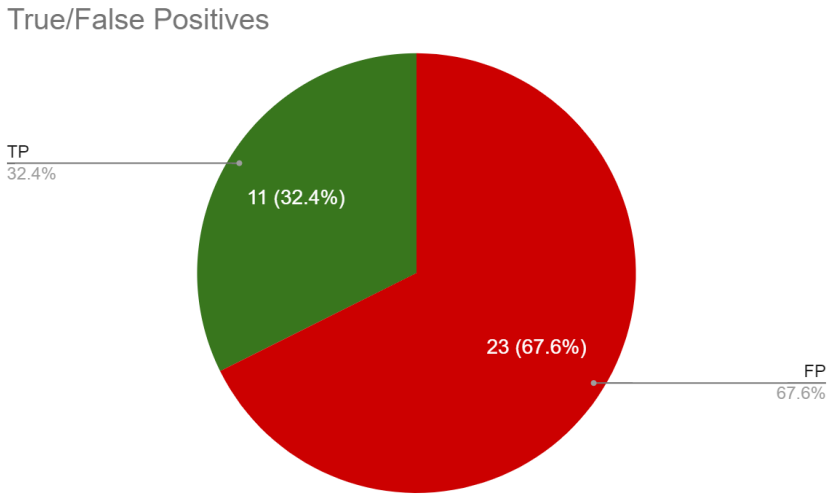


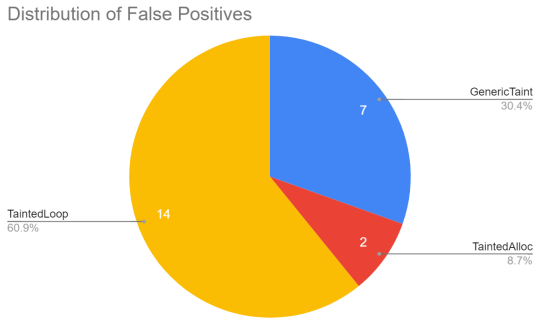


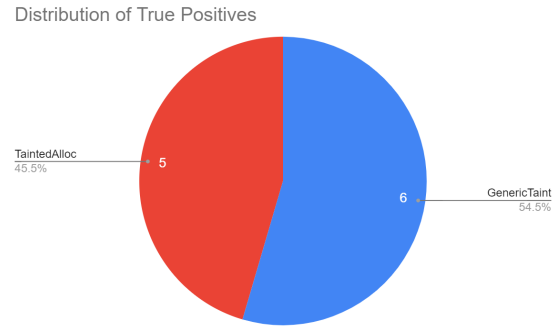


Additional results with spread mode









https://github.com/dkrupp/llvm-project/tree/spread_taint

Future work

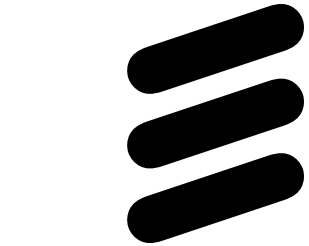
ERICSSON

- **Analysis of library interfaces**
  - Today, the taint sources can only be called functions (e.g. scanf, fopen, etc.)
  - How could I scan a library with interface functions with incoming tainted parameters?
  - Extending the configuration with a possibility to define tainted top-level function
  - Functions declared in interfaces

- **Taint analysis through C++ virtual functions across Translation Unit**

  Work by Balázs Benics:
  Attempting Cross Translation Unit Taint Analysis for Firefox
  https://attackanddefense.dev/2025/12/16/attempting-cross-translation-unit-static-analysis.html

  The goal is to perform taint analysis with the Clang Static Analyzer, with an attempt to resolve virtual functions with CTU.



ERICSSON

Appendix - Evaluation settings on open-source projects

ERICSSON

- The Environment was considered untrusted (Env variables, files, network, ...)
  - This may be too pessimistic…
  - There were wrapper tools (which intentionally called execl(..))
- Application test code was also included
- If the programmer implemented defense against the attack, but it was undetected by the analyzer, the result was considered false positive