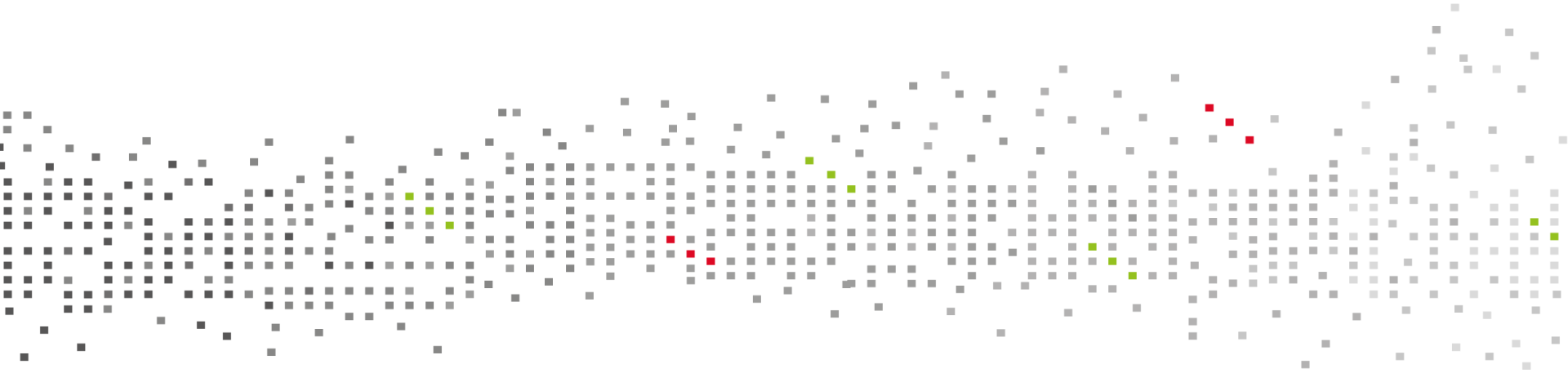


FINDING SECURITY VULNERABILITIES BY CODE AUDIT & STATIC ANALYSIS

JAIF | David Féliot (CEA Leti) | 23 May 2019

FINDING SECURITY VULNERABILITIES

OpenSSL
Heartbleed

Discovered

Code audit (manual static)
"laborious auditing, line by line"
+ static & dynamic techniques?

Fuzzing (automated dynamic)

More reliable?
More efficient?

Functional-level weaknesses?

Post analysis:

Not detected

Detected

False alarm?

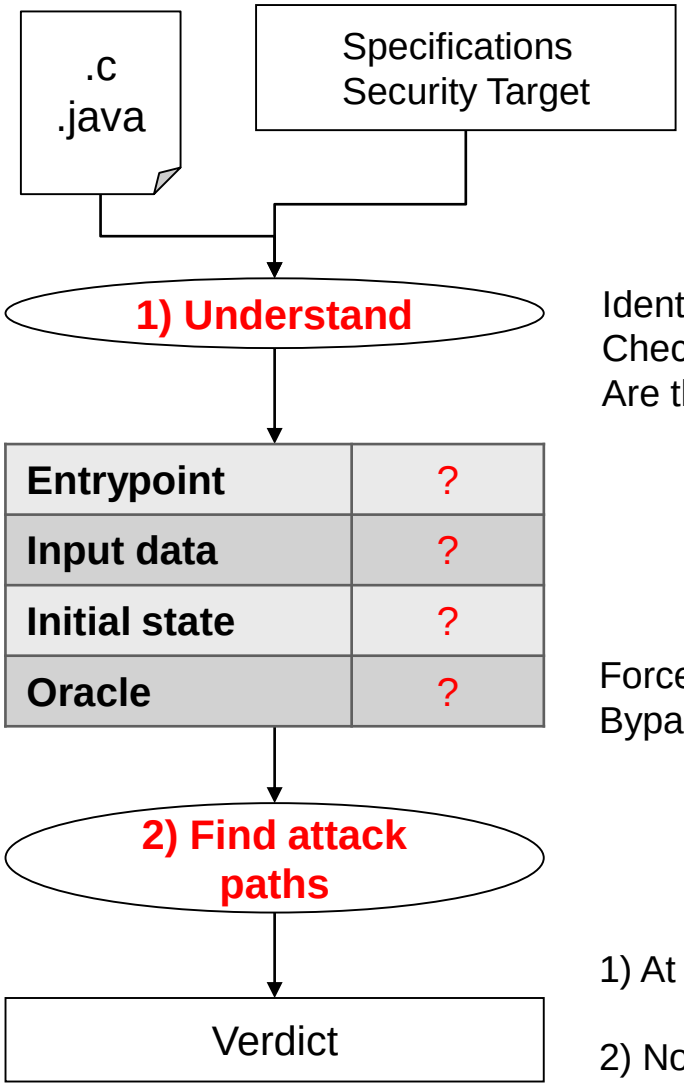
Automated static:

Checkers
Data flow (user input → target) + Symbolic Execution
Unsound detection of many weaknesses (CWE)

Sound static analysis
Verified property: no "runtime error"
(CWE subset, e.g. Buffer Over-read)

frama C Software Analyzers
Eva

2-STEP METHOD



Code audit + value analysis

Identify: user inputs, sensitive functions and data (assets)
Check the paths that are specified
Are there any unexpected paths that could be exploited?

Forced security function (e.g. verify PIN, access control)
Bypassed security function (e.g. Java Card firewall)

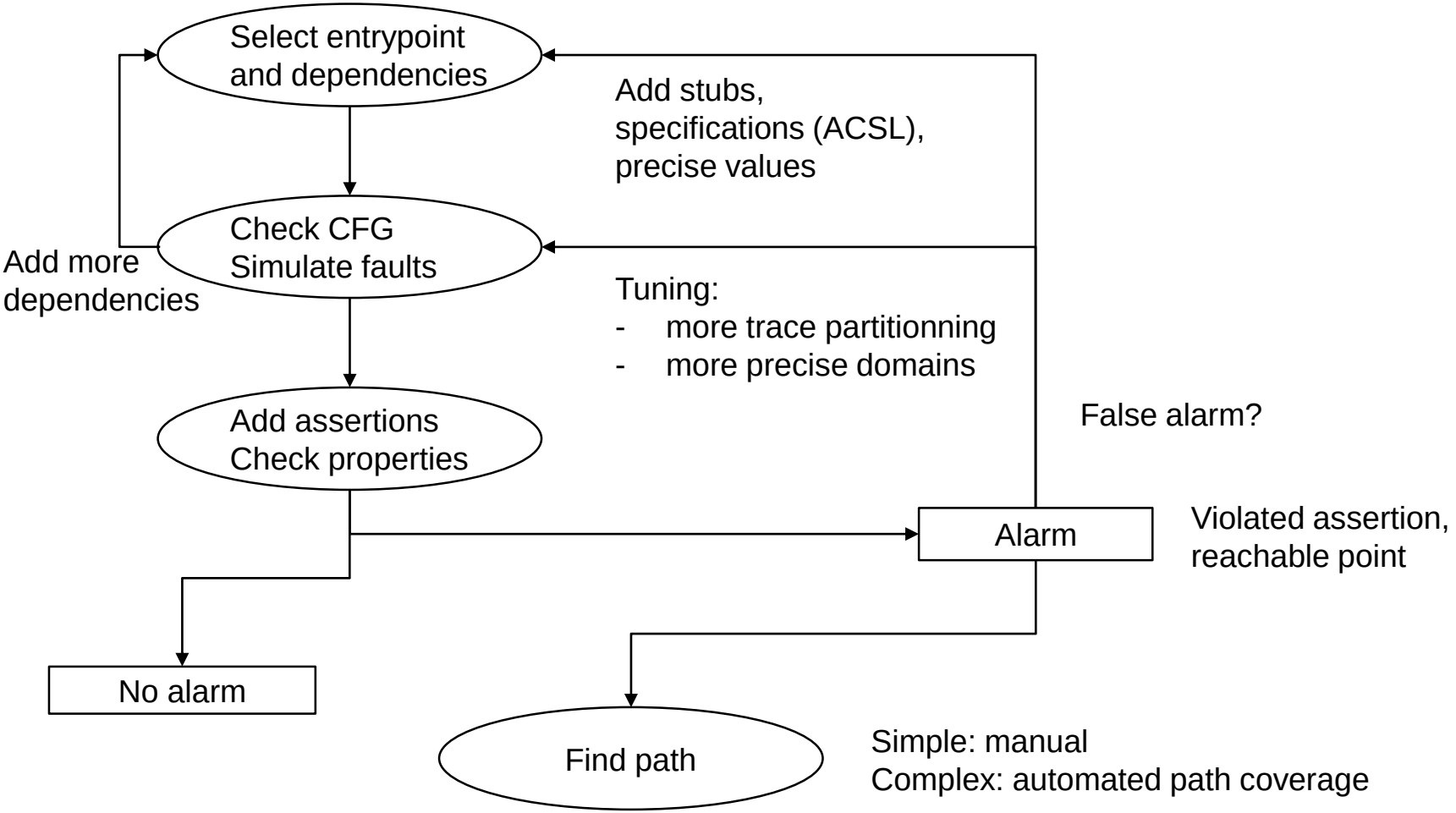
Automated path coverage

1) At least one attack path found

2) No attack path found: all-path coverage?
True: secure; False: unknown

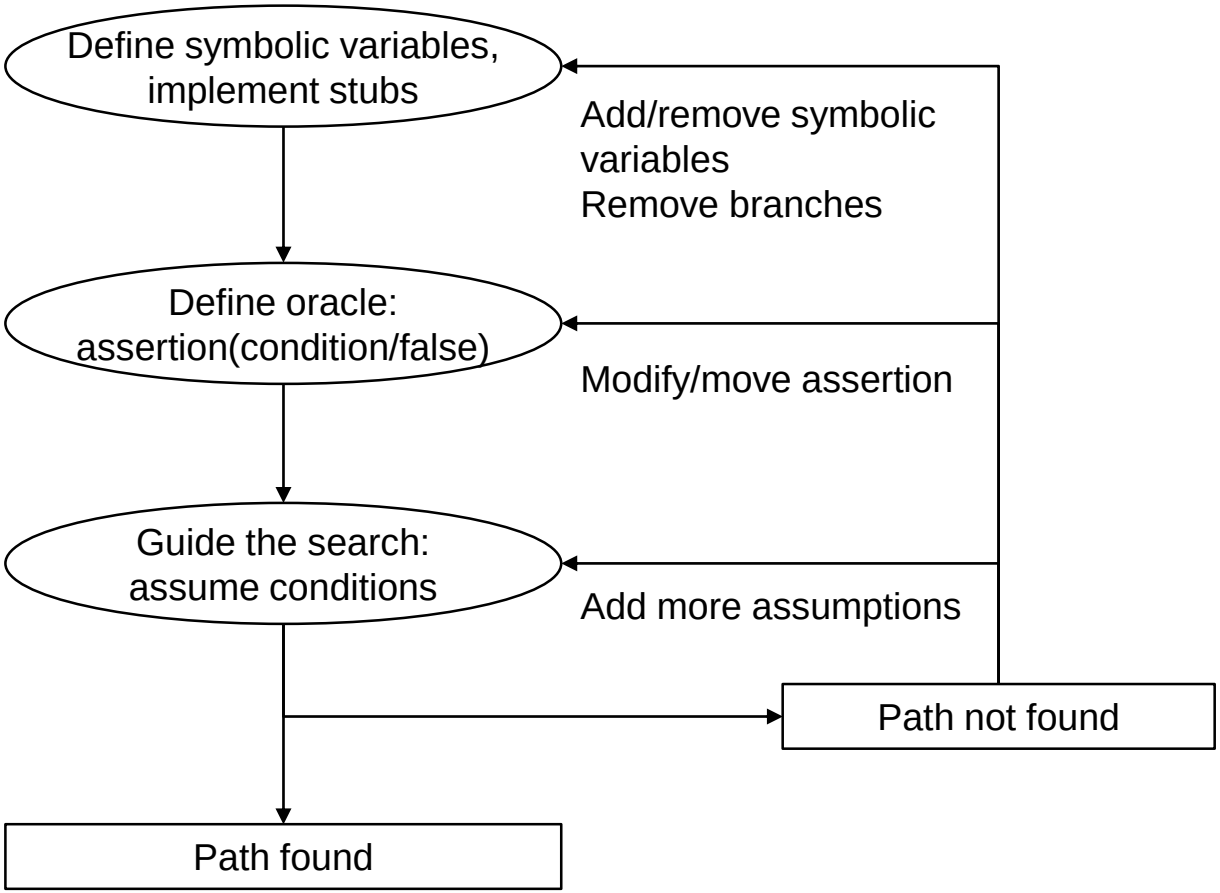
CODE AUDIT + VALUE ANALYSIS

Abstract interpretation with Frama-C Eva: abstract domains + trace partitionning



AUTOMATED PATH COVERAGE

Dynamic Symbolic Execution with Frama-C PathCrawler, and KLEE



EXAMPLE: BASH SHELLSHOCK

Only one function definition should be allowed in an environment variable:

```
setenv("f", "(){echo \"…\";}")
```

1) Should not allow a command that is not a function definition:

```
setenv("f", "echo \"…\"")
```

2) Should not allow batch commands:

```
setenv("f", "(){:;};echo \"…\"")
```

```
int parse_and_execute(char *string, const char *from_file, int flags) {
  ...
  with_input_from_string(string, from_file);
  ...
  while (*(bash_input_string)) {
    if (parse_command() == 0) {
      ...
      last_result = execute_command_internal(command);
      ...
    }
  }
  return last_result;
}
```

Multiple commands are parsed

```
int execute_command_internal(COMMAND *command) {
  switch (command->type) {
  case cm_simple:
    ...
    break;
  case cm_for:
    ...
    break;
  case cm_function_def:
    ...
    break;
  ...
  }
  ...
}
```

Potentially reachable command types

Really reachable points?
Caused by missing dependencies?
Caused by approximation?
→ Make the behaviour more precise and check the CFG

```
void initialize_shell_variables (char **env, int privmode) {
  for (string_index = 0; string = env[string_index++]; ) {
    ...
    if (... && STREQN ("() {", string, 4)) {
      ...
      parse_and_execute (temp_string, name, flags);
      ...
```

Filter function definitions

STUBS & PRECISE VALUES

```
void with_input_from_string(string, name)
  char *string;const char *name; {
  bash_input_string = string;
}


COMMAND parsed_commands[2];
int command_idx;


int parse_command() {
  if (command_idx < 2) {
    command = &parsed_commands[command_idx++];
    return 0;
  } else {
    bash_input_string = "\0";
    return 1;
  }
}
```

Stubs

```
int main() {
  parsed_commands[0].type = cm_function_def;
  parsed_commands[1].type = cm_simple;
  command_idx = 0;
  char *string = "() { :;}; echo vulnerable";
  parse_and_execute(string, from_file, flags);
```

Precise values

```
int parse_and_execute(char *string, const char *from_file, int flags) {
  ...
  with_input_from_string(string, from_file);
  ...
  while (*(bash_input_string)) {
    if (parse_command() == 0) {
      ...
      last_result = execute_command_internal(command);
      ...
    }
  }
  return last_result;
}
```

Loop unrolling

&parsed_commands{[0], [1]}

```
int execute_command_internal(COMMAND *command) {
  switch (command->type) {
  case cm_simple:
    ...
    break;
  case cm_for:
    ...
    break;
  case cm_function_def:
    ...
    break;
    ...
  }
  ...
}
```

Potentially reachable

CHECK PATCHED VERSION

```
void with_input_from_string(string, name)
  char *string;const char *name; {
  bash_input_string = string;
}


COMMAND parsed_commands[2];
int command_idx;


int parse_command() {
  if (command_idx < 2) {
    command = &parsed_commands[command_idx++];
    return 0;
  } else {
    bash_input_string = "\0";
    return 1;
  }
}
```

Not precise

```
int main() {
  //parsed_commands[0].type = cm_function_def;
  //parsed_commands[1].type = cm_simple;
  command_idx = 0;
  char *string = "() { :;}; echo vulnerable";
  parse_and_execute(string, from_file, flags);
```

```
int parse_and_execute(char *string;const char *from_file;int flags) {
  ...
  with_input_from_string(string, from_file);
  ...
  while (*(bash_input_string)) {
    if (parse_command() == 0) {
      ...
      if ((flags & SEVAL_FUNCDEF)
        && command->type != cm_function_def) {
        break;
      }
      ...
      last_result = execute_command_internal(command);
      ...
      if (flags & SEVAL_ONECMD)
        break;
    }
  }
  return last_result;
}
```

&parsed_commands{[0]}

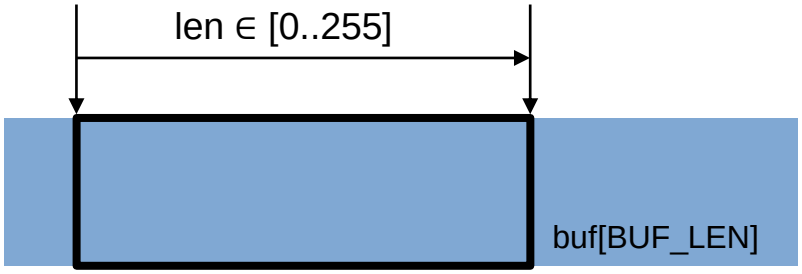
```
int execute_command_internal(COMMAND *command) {
  switch (command->type) {
  case cm_simple:
    ...
    break;
  case cm_for:
    ...
    break;
  case cm_function_def:
    ...
    break;
    ...
```

Not reachable (proved)

NO RUNTIME ERROR

Example: buffer overflow

offset ∈ [0..65535] offset + len <= BUF_LEN




```
if (offset + len <= BUF_LEN)
  for (int i = 0; i < len; i++)
    buf[offset++] = …
```

Gauge: offset = <initial offset> + i
Loop invariant: offset – i + len <= BUF_LEN and i < len → offset < BUF_LEN

Different precision levels:

- Interval — offset ∈ [0 .. 65535]
- Interval + trace — offset ∈ [0 .. BUF_LEN + 255 - 1]
- Gauges — offset ∈ [0 .. BUF_LEN + 255 - 1]
- Octagon — offset ∈ [0 .. 65535] (3-var invariant not inferred)
- Polyhedra (strict or loose) — offset ∈ [0 .. BUF_LEN - 1]

Precise value ‘len’:

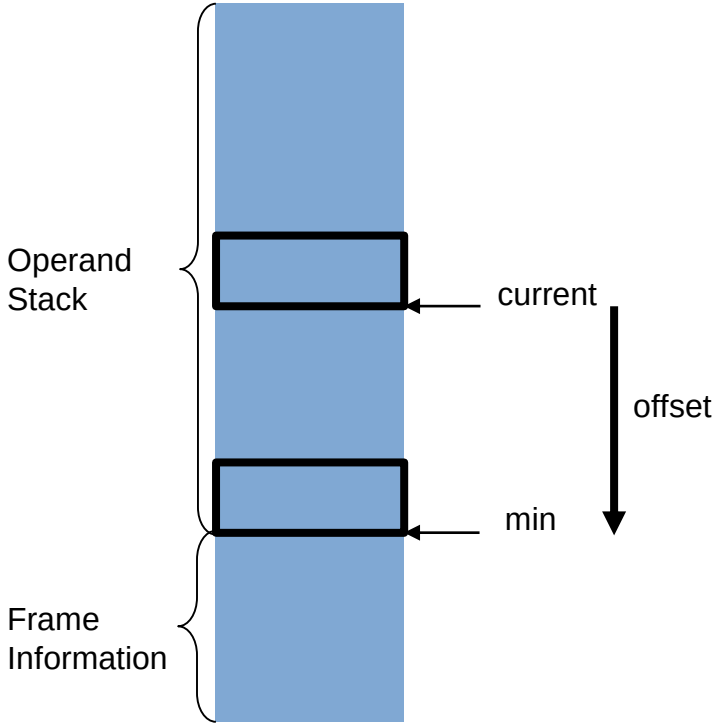
- Interval + trace — offset ∈ [0 .. BUF_LEN - len + len - 1]

NO UNEXPECTED BEHAVIOUR

Example on a Java Card platform:
"Manipulating the Frame Information With an Underflow Attack"
Emilie Faugeron (CARDIS 2013)

```
if (current - offset < min) return;
…
…
copy_current(offset);
…
```

```
void copy_current(uint offset) {
  //@ assert current - offset >= min;
  …
  buf[current - offset] = buf[current];
  …
}
```



NO UNEXPECTED REACHABLE POINT

Example: ~ ShellShock + complex path

```
char buf[BUF_LEN];
command_t cmd;
int offset;
char mode;

void handle_command() {
  cmd.type = 0;
  cmd.op = 0;
  read_type();
  read_op();
  switch (cmd.op) {
  case 1:
    op1();
    break;
  case 2:
    op2();
    break;
    ...
  }
}
```

```
int main() {
  offset = 0;
  buf[0] = 0x0A;
  handle_command();
  return 0;
}
```

'mode' value is not precise

```
void read_type() {
  char id = buf[offset++];
  if (mode == 0) {
    if (id == 0x0A) {
      cmd.type = 1;
    } else if (id == 0x0B) {
      cmd.type = 2;
    }
  }
}

void read_op() {
  if (cmd.type == 1) {
    char id = buf[offset++];
    if (id == 0x0B) {
      extended_read_op();
    } else {
      cmd.op = 1;
    }
  } else if (cmd.type == 2) {
    cmd.op = 2;
  }
}
```

Complex path and input data 'buf': 0x0A 0x0B 0x0C…

```
void extended_read_op() {
  char id = buf[offset++];
  if (id == 0x0C && mode == 1) {
    cmd.op = 2;
  }
}
```

Actually not reachable (merged states)

SIMULATE FAULT INJECTION

Simulate the impact of faults on the CFG

Two fault models:
- test inversion
- faulted value

Applied:
- manually when checking the CFG
- automatically with Lazart

```
read_access_level = VERIFY_PIN;

int res = access_control(READ);

LAZART_ORACLE(res == 1);
```

```
// stub
int pin_is_validated() {
  return 0;
}
```

```
int parse_and_execute(char *string;const char *from_file;int flags) {
  ...
  with_input_from_string(string, from_file);
  ...
  while (*(bash_input_string)) {
    if (parse_command() == 0) {
      ...
      if ((flags & SEVAL_FUNCDEF)
        && command->type != cm_function_def
        && ! (command_idx == 2 && FAULT_2)) {
        break;
      }
      ...
      last_result = execute_command_internal(command);
      ...
      if (flags & SEVAL_ONECMD
          && ! (command_idx == 1 && FAULT_1))
        break;
    }
  }
  return last_result;
}
```

```
int execute_command_internal(COMMAND *command) {
  switch (command->type) {
  case cm_simple:
    ...
    break;
  case cm_for:
    ...
    break;
  case cm_function_def:
    ...
    break;
```

Potentially reachable

CONCLUSION

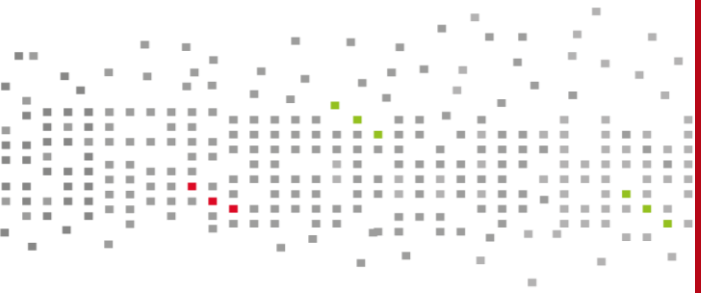
Code audit + value analysis:
- understand code from manually selected entrypoints
- simulate fault injection
- check properties (no runtime error, no unexpected behaviour)

Find potential vulnerabilities

Prove that a vulnerability has been fixed

Automated path coverage: find a complex path reaching a vulnerability with or without fault injection

Tools currently used:
- Value analysis: Frama-C Eva
- DSE: Frama-C PathCrawler, KLEE, Lazart
- Java Card to C: Frama-C JCard
- C++ to C: Frama-Clang



Leti, technology research institute
Commissariat à l'énergie atomique et aux énergies alternatives
Minatec Campus | 17 avenue des Martyrs | 38054 Grenoble Cedex | France
www.leti-cea.com