POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Security-oriented dynamic code analysis in automotive embedded systems

Relatori
Prof. Gianpiero Cabodi
Dott. Fabrizio Sebastiano Finocchiaro

Candidato
Daniele Montisci

Anno Accademico 2017-2018

Contents

List of Tables

List of Figures

Chapter 1

Motivations and objectives

Dynamic code analysis[1] is a set of techniques for computer security analysis based on monitoring of running code, which allows to exploit run-time information. In this work, two among these techniques, dynamic taint analysis and symbolic execution, are taken into consideration.

White-box testing is a method for testing a software application by exploiting the knowledge of its source code. In the past, it was used almost exclusively at unit level; recently, it has started to be used also at integration and system levels. Traditional approaches to white-box testing require a significant effort by the tester. It is necessary to manually inspect and to understand precisely the inner working and structures of the software module under test in order to reason out an exhaustive set of test cases. Sometimes it is not feasible to figure out all the admitted behaviours due to the complexity of the code, a part of which may thus remain untested. Furthermore, the obtained test cases set is closely related to the implementation details and not to the functional specifications of the module. This is problematic when a functionality needs to be re-implemented, because also the corresponding test cases set must be re-created. Otherwise, it is possible to incur in false negatives (non existing errors) and/or false positives (non triggered errors). All of this impacts on the development cost and time.

The necessity to lighten this effort and at the same time to improve the quality of the generated test cases set, has brought to the introduction of symbolic execution (SE), described in Chapter 2. Its main application is the automatic generation of a set of test cases given the source code or the binary of the application under test. This set is virtually capable of exploring all the admitted behaviours, intended or not, of an implementation. The key idea of the technique consists in using symbolic values, capable of representing any concrete value, as input and in representing internal and output values as symbolic expressions over the input values. Every execution path passes through the true or the false branches of a sequence of conditional statements and it is described with a set of constraints over the input values. By solving them, the corresponding test case is obtained. The main limitations of SE are the path explosion and the constraints solving, which prevented it from being applicable for the testing of real pieces of software. Recently there has been a renewed interest caused by the advancement in constraint solving techniques and by the introduction of the possibility to intermix symbolic with concrete execution. This has made SE suitable especially for testing embedded software, whose typical size ranges from hundreds of thousands to few millions lines of code. Being frequently critical and not continuously upgradable, it requires thorough testing before deployment.

Dynamic taint analysis (DTA), described in Chapter 3, allows to observe the information flow during the execution of a software component. Its main application is the verification of security properties. It consists in marking data incoming from a certain source with a taint, consisting in shadow bits, and in observing if this taint can be propagated to a certain destination. Propagation is performed during the execution of base computations (e.g. additions, multiplications, ...): if at least one of the operand is marked, then the same taint is assigned to the result. If the source is considered safe and the destination unsafe, then it is possible to verify the confidentiality of sensitive information by checking that the taint cannot propagate. Otherwise, if the source is unsafe and the destination is a trusted memory location or register, it is possible to verify the integrity of the destination, whose sensitive information must not be under the influence of data potentially injected by an attacker, again by checking the impossibility for the taint to propagate.

Currently, SE and DTA are considered to be among the most promising techniques for dynamic code analysis. The objective of this work is to verify their applicability in the testing phase of real software systems, specifically embedded software in automotive context. An electronic control unit operating inside a car has been chosen as use case. This unit provides two main functionalities: filtering and routing all the messages exchanged over CAN buses during the normal vehicle operation and authentication of an external diagnostic device, whose messages are allowed to pass in case of success. A detailed functional description of the unit and its firmware architecture are reported in Chapter 6. In order to achieve this objective, some publicly available tools have been taken into consideration. Those that best suited the kind of analyses to be performed on the use case have been selected.

In order to deal with the limitations of SE, a template procedure for creating a test harness, making the analysis feasible and obtaining meaningful results has been defined; it is discussed in Chapter 7.

SE is affected by the so-called path explosion problem, consisting in the repeated forking of the execution flow in correspondence of loops and recursions whose number of iterations depend on symbolic values. In order to control this problem, it is necessary to be able to constrain adequately the input data that have been made symbolic. The approach that has been judged more effective is called source code instrumentation and it is adopted by the KLEE tool and its derivative S2E, which are described in Chapter 4.

All the freely available tools for DTA implement the traditional approach described above. When scaled to real systems, it turns out to be insufficient, due to the amount of false negatives and false positives yielded. The adopted solution consisted in implementing a recent algorithm that interprets DTA as an application of SE. Multi-path DTA makes possible to have virtually exact results or, if the path explosion problem becomes unmanageable, to control more effectively the sources of false negatives and false positives. A tool based on the traditional approach and the prototype implementing the aforementioned algorithm are described in Chapter 5.

Chapter 8 and Chapter 9 detail the analysis procedure on the use case. The former focuses on the messages filtering/routing functionality. The latter focuses on the external diagnostic devices authentication procedure. For each module or set of modules under test, first the choices in the construction of the test harness are discussed, then the obtained results are reported. Chapter 8 reports a comparison between DTA based on SE and the traditional approach, proving the powerfulness of the former. Since the authentication procedure is affected by some vulnerabilities, Chapter 9 lists the affected portion of code and the mechanism for each bug-triggering input.

Chapter 2

Symbolic execution

Symbolic execution[1][2] is a technique consisting in substituting concrete data values given as input to a program with symbolic values, in representing (part of) the values of program variables as symbolic expressions over these symbolic values, and in expressing the output values of the program as a function of the inputs. This allows us to explore the behaviour of a program (i.e. to test it) on many possible input at once by implicitly enumerating them. In the strict sense, a symbolic value is unconstrained, which means that it can represent any possible concrete value. A program variable can take a simple symbolic value, or in the general case a symbolic expression, also called constrained symbolic value (using a less precise but more intuitive terminology), that limits the possible concrete values that it can assume. A feasible execution path is defined by the sequence of branches (then or else) taken at each encountered conditional statement. All the feasible execution paths of a program can be represented with an execution tree, which forks in correspondence of each conditional statement. To execute symbolically a program it is necessary to maintain for each execution path a symbolic state (symbolic expressions assumed by variables) and a path constraint (conjunction of the conditions of the encountered conditional statements). An execution path is feasible if its path constraint can be solved, yielding a concrete sample value. At the end of the execution of a feasible path, a test case (i.e. a set of input values) that covers it can be generated. When a conditional statement is encountered, both the symbolic state and the path constraint are forked: the first path constraint is incremented with the condition (*then* branch), the second with the negation of the condition(*else* branch). Both branches can be taken if the corresponding path constraints are feasible (at least one is, if the parent path is feasible).

In Listing 2.1 there are 2 conditional statements and 3 possible execution paths, one of which is infeasible: the *assert(false)* statement is not reachable, whatever value the variables x and y take. The corresponding graph of the execution paths is reported in Figure 2.1.

```
int x, y;
... // Give the input variables "x" and "y" symbolic values "X"
    and "Y" respectively.
if (x > y) {
     x = x + y;
     y = x - y;
     x = x - y;
     if (x - y > 0) {
          assert(false);
     }
}
```

Listing 2.1. Basic symbolic execution example.

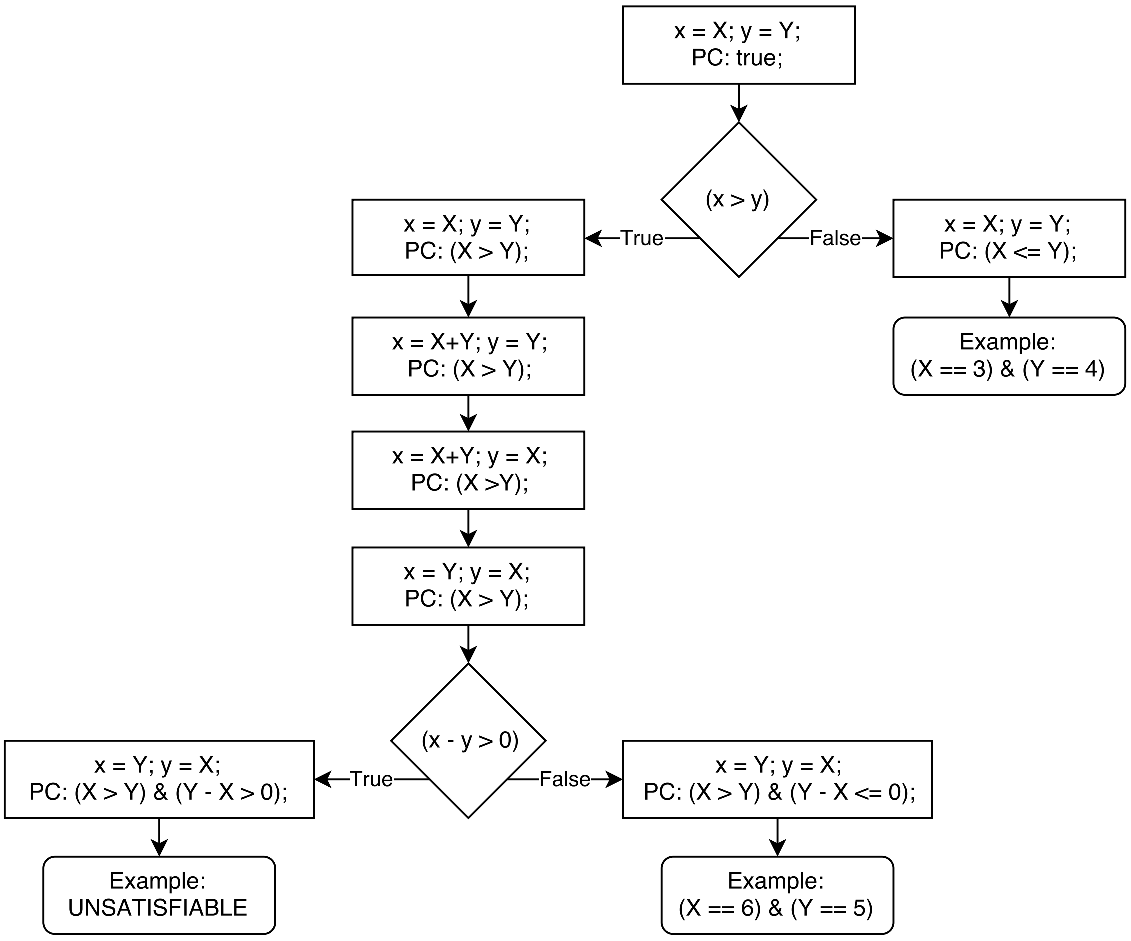

Figure 2.1. Execution tree of code example in Listing 2.1.

2.1 Challenges

- The symbolic memory address problem arises when trying to de-reference a symbolic value, derived from user input: a possible solution is to dereference any possible value (explicit enumeration), which requires to fork a new execution path for each one of them[3].
- The symbolic jump problem is related to the previous one, it arises when trying to jump to a symbolic location: a possible solution is to use static analysis to determine all the possible jump targets.
- The path selection problem consists in deciding which branch to follow first when a conditional statement is encountered. This is particular important in case of never-ending loops, in which the symbolic execution gets stuck, preventing the exploration of other parts of the program. Possible strategies include DFS search with a limit on the number of iterations, concolic testing (see 2.2), random selection (favours shallows states).
- External libraries not instrumented for symbolic execution and OS system calls can abort the analysis, because it is not possible to symbolically explore them: concolic testing (see 2.2) offers a partial solution, but it is also possible to emulate system calls so that they return symbolic data while keeping track of their side effects[4].

2.2 Modern symbolic execution

Modern techniques for symbolic execution[2] mix concrete and symbolic execution.

- Concolic (concrete and symbolic) testing maintains a symbolic state, that maps only variables that have a symbolic value, and a concrete state, that maps all variables to their concrete value. As consequences, a variable can have at the same time a concrete and a symbolic value, and concrete input values are needed. While the program under testing is executed, the constraints on the input values posed by the encountered conditional statements are gathered and solved, so that at the next execution an alternative feasible execution path is selected. This process iterates until all feasible execution paths are executed, or a user-defined criteria is met.
- Execution-generated testing (EGT) maintains a symbolic and a concrete state similarly to concolic testing, but the way in which concrete and symbolic execution are mixed is different: if an instruction uses only concrete operand values, it is executed concretely (potentially at native speed), otherwise (i.e. at least one symbolic operand value) symbolically.

2.2.1 Pros and cons

Pure symbolic execution cannot explore parts of the program under test that interact with an external library not instrumented for symbolic execution or that issue an OS system call. EGT can execute such a call concretely if all the argument values are concrete, otherwise if at least one is symbolic, it needs to concretize them (i.e. to solve the path constraint built so far). Concolic execution can use the concrete values (derived from the input values) to perform the call. This constraint simplification allows to execute all the program under test, but the analysis may be incomplete: some feasible execution path may be left unexplored, thus the coverage of the test input set may be reduced.

2.3 Limitations

- The main limitation of symbolic execution is the path explosion problem[2]: when executing loops or recursive procedures the number of feasible paths can grow huge if the termination condition depends on symbolic input data. It is necessary to put an heuristic limit on the number of forked paths or on the number of iterations. The number of execution paths, in the worst case, is exponential on the number of static branches in the code. The problem is mitigated by the fact that not all branches may be feasible and not all conditional statements have a condition dependent on symbolic input data. The problem can be addressed by using search heuristics for prioritizing path exploration, by pruning or merging redundant paths (i.e. paths with the same path constraint), ...

```
1 unsigned int buffer[SIZE];
2 unsigned int num;
3 ... // Give input variable "num" a symbolic value "N".
4 if (num <= SIZE) {
5     for (unsigned int i = 0; i < num; i++) {
6         buffer[i] = i;
7     }
8 }
```

Listing 2.2. Example of code causing path explosion.

A piece of code causing path explosion is reported in Listing 2.2, and the corresponding execution tree starting from the for loop in Figure 2.2: only the input variable *num* is given a symbolic value, which in the true branch if the if-then-else statement starting from line 5 can vary from 0 to *SIZE*, extremes included; before executing the body of the for loop the termination condition is checked and since both branches are feasible until the index becomes equal to *SIZE*, a new execution path is forked and the relative test case is created for each possible concrete value that the variable *num* can take; the resulting number of execution paths is equal to *SIZE* + 1 and, if *SIZE* is set to a static value in the order of one hundred of thousands or one million, even the analysis of such a simple piece of code poses feasibility problems (i.e. execution time and memory footprint) with the symbolic execution tools taken into consideration.

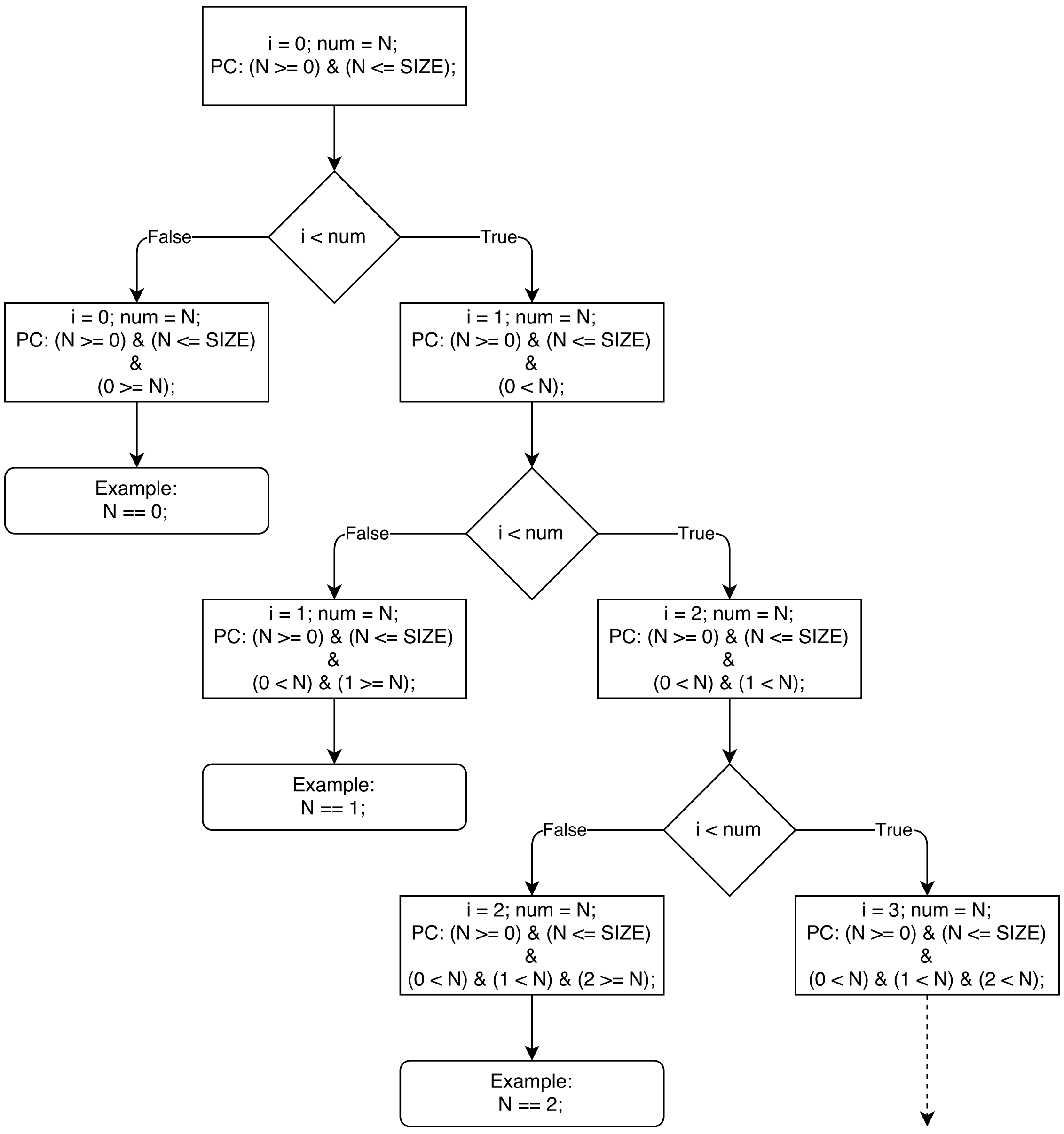
i = 0; num = N;
PC: (N >= 0) & (N <= SIZE);
i < num
False
True
i = 0; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 >= N);
i = 1; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 < N);
Example: N == 0;
i < num
False
True
i = 1; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 < N) & (1 >= N);
i = 2; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 < N) & (1 < N);
Example: N == 1;
i < num
False
True
i = 2; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 < N) & (1 < N) & (2 >= N);
i = 3; num = N;
PC: (N >= 0) & (N <= SIZE) & (0 < N) & (1 < N) & (2 < N);
Example: N == 2;

Figure 2.2. Execution tree of code example in Listing 2.2.

- Some path constraints can be impossible or hard to solve (for example non-linear ones): this may preclude or slow down the exploration of feasible execution paths, and thus reduce the coverage of the yielded test-cases set. The problem can be addressed by eliminating irrelevant constraints (if the constraint of a branch depends on a subset of the variables inside the path constraint, then only the corresponding subset of constraint is needed to determine the feasibility of the execution path) and by incremental solving (a solution for a set of constraints is a solution also for a subset of them, and usually also for a superset).

Chapter 3

Dynamic taint analysis

Dynamic taint analysis[1] (DTA), also known as dynamic information flow analysis, is a technique consisting in marking with taints some data sources, according to some taint introduction rules, and tracking the propagation of the taints, according to some taint propagation rules. Taints may consist in simple binary values or in arrays of bits[5].

- A taint introduction rule specifies how to introduce taints into the system. A typical rule is to initialize all registers and memory cells as untainted, and to specify that some system calls or some functions from a certain library always return tainted values. Some tools[5] allow also to arbitrarily mark variables as tainted.

- A taint propagation rule specifies, for each operation, the tainting of the result according to the tainting of the operands. A typical rule is to taint the result of an operation if at least one of the operands is tainted. This rule can be extended to the case in which we have multi-bit taints of the same length, using a bitwise-or.

DTA is subject to two errors: over-tainting and under-tainting. Their occurrence is influenced by the taint propagation rules adopted. A DTA system that does not suffer over-tainting and under-tainting is defined precise.

3.1 Taints propagation

Taint values can be associated with data values (i.e. the contents of memory cells and of CPU registers, with memory addresses (both to data and to instructions) and with the control-flow (i.e. with the program counter). These different types of taints may or may not be propagated according to the adopted propagation rules.

3.1.1 Data-flow taints

To propagate the data-flow taints consists in considering only simple assignments, unary and binary operators (arithmetical, logical, ...) and in assigning a combination of the taint of the operands to the result. Considering the example in Listing 3.1, when the content values of variables *a* and *b* are summed, in parallel the taint values are combined with a bitwise OR operator. Both these results are assigned to variable *c*. The same applies for variable *e* which, at the end of the execution, has the taint value of *a*. The propagation is depicted in Figure 3.1.

```
int a = 1; int b = 4; int d = 2; int c, e;
... // Assign variable "a" taint value 1.
c = a + b;
e = c * d;
```

Listing 3.1. Since a is tainted, "c" and then "e" will be tainted.

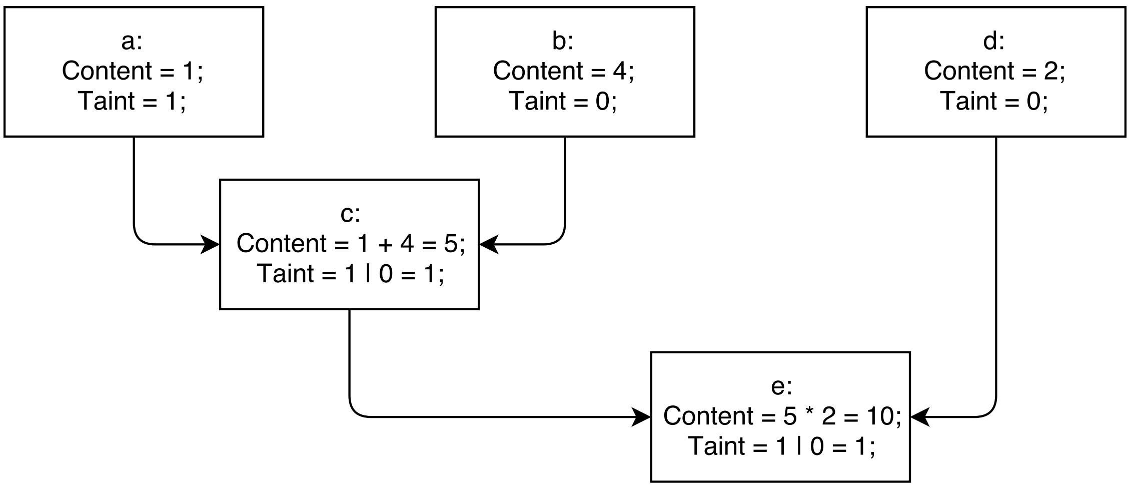

Figure 3.1. Data-flow taint propagation in Listing 3.1.

3.1.2 Addresses taints

If an array is accessed using a tainted index, or if a tainted value is dereferenced, the result of the memory operation is tainted. To achieve this, it is necessary to propagate the taints of both values and addresses, otherwise under-tainting occurs. Considering the example in Listing 3.2, when variable *b* is used to index the array *a*, the address of the first cell of *a* and the content value of *b* are summed, while the taint values of the seventh cell of *a* and of *b* are combined. Both these results are assigned to variable *c*. The propagation is depicted in Figure 3.2.

```
char a[SIZE]; int b = 4; char c;
a[b] = 6;
... // Assign variable "b" taint value 1.
c = a[b];
```

Listing 3.2. Since "b" is tainted, "c" will be tainted.

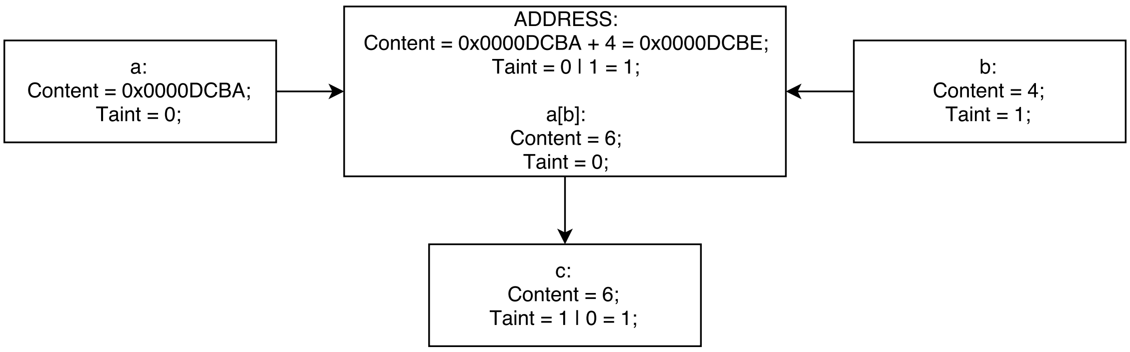

Figure 3.2. Address taint propagation in Listing 3.2.

3.1.3 Control-flow taint

It is possible to propagate the control-flow taint in addition to the data-flow taints: if the condition of a branch or the address of an indirect branch are tainted, then all the variables assigned inside both the true and false branch bodies are potentially tainted. If inside one of the branch bodies a variable is assigned a value differing from the one it contained before the branch and, at the same time, differing from the one it is assigned inside the other branch body (if present), then it will be tainted. In Listing 3.3, variable *b* is assigned value 3 only in the true branch, a value different both from the original one, which was 0, and from the one it is assigned inside the false branch, again 0, as shown in Figure 3.3. The same applies for variable *c*. There may be pieces of code after conditional branches whose execution does not depend on the condition, for example the assignment to variable *d*, which will not be tainted: this might seem banal, but while when considering the source code the bodies of the branches are easily identified, when considering the corresponding binary they are not, and all the tools considered in this work perform dynamic taint analysis on binaries.

```
int a = 5; int b = 0; int c = 0; int d = 0;
... // Assign variable "a" taint value 1.
if (a == 5) {
    b = 3;
} else {
    c = 4;
}
d = 6;
```

Listing 3.3. Since "a" is tainted, "b" and "c" will be tainted, but "d" will not.

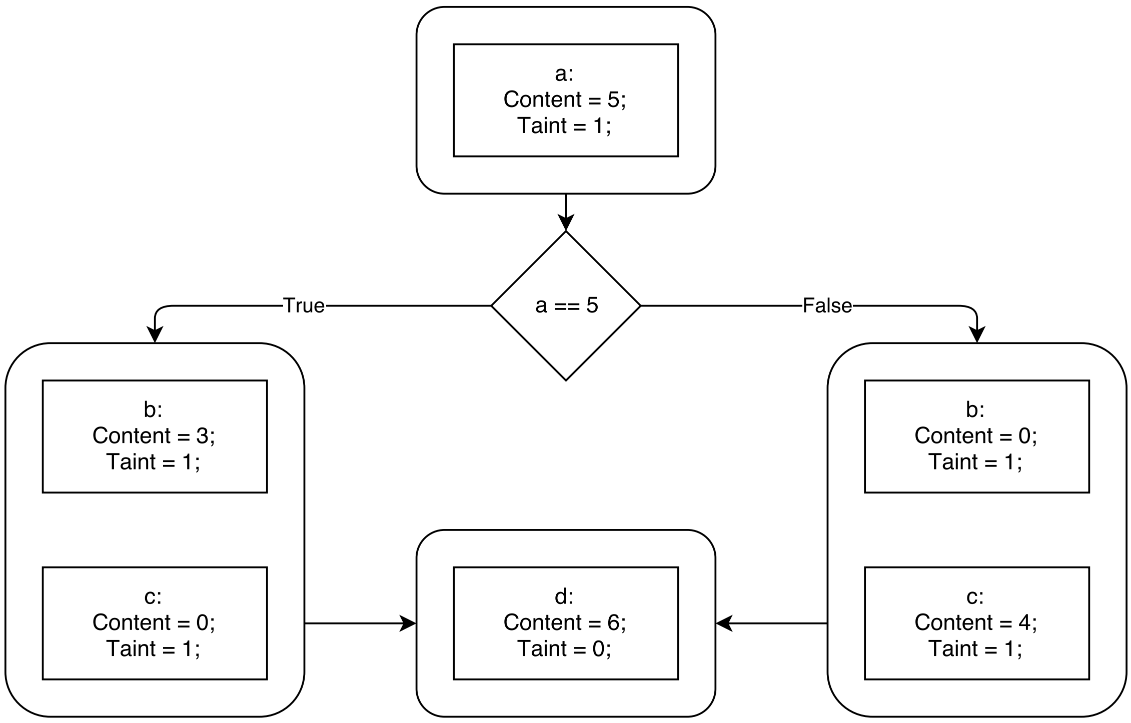

Figure 3.3. Control-flow taint propagation in Listing 3.3.

It is possible to deduct that control-flow taint propagation requires taking into consideration both branches of a conditional statement; executing only one while completely disregarding the other one as it happens in an ordinary program execution may make the analysis imprecise. In the general case, computing the control dependencies requires a static analysis or a multi-path dynamic analysis of the code, a pure single-path dynamic analysis incurs in under-tainting (do not track control-flow taint) or over-tainting (once the control-flow is tainted, it cannot be un-tainted). To achieve a precise DTA, it is required to track data-flow taints, addresses taints and control-flow taint.

3.2 Taints removal

Taints can be removed when the program computes constants from tainted values. In Listing 3.4, variables b and c are always equal to 0, independently of the value of the operand a, thus they will not be tainted. The *XOR* case is critical, because it is often used to clear x86 registers. Recognizing these situations is called the taint sanitization problem.

```
int a = 5; int b = 0; int c = 0;
... // Assign variable "a" taint value 1.
b = a ^ a;
c = a - a;
```

Listing 3.4. "a" is tainted, but "b" and "c" will not.

3.3 Limitations

The purpose of dynamic taint analysis is to determine if a taint source can influence the value of a taint sink. As discussed in Section 3.1, traditional techniques apply the data-flow taints propagation logic associated to an instruction only when that instruction is executed, and perform static analysis of the code to determine the propagation of the control-flow taint. There are a number of situations in which this approach fails:

- A taint source influences a taint sink but the taint propagation logic cannot be applied, because the control-flow prevents the execution of the corresponding instruction: the analysis suffers from under-tainting. In Listing 3.5, variable a influences b, because depending on its value b may take content value 2 or 3, but in the execution path chosen by the control flow the variable assignment in line 4 is not executed. The KLEE-TAINT[5] tool fails in recognizing this influence and determines that b is not tainted.

```
1 int a = 1;  // Taint source - Tainted
2 int b = 2;  // Taint sink - Not tainted
3 if (a == 5) {
4       b = 3;  // Application of taint propagation logic.
5 }
```

Listing 3.5. Example of under-tainting caused by an instruction not being executed.

  The DYTAN[6] tool, in order to deal with such cases, performs an initial static analysis of the code in order to determine which variables are assigned inside the body of at least one branch of each conditional statement: if a variable is assigned only inside one branch, then DYTAN inserts a self-assignment of the same variable inside the other branch. The purpose is to have same set of assigned variables in both branches, so that when the dynamic part of the analysis is performed, the control flow taint can be effectively propagated during the execution of condition-dependent variable assignments. In Listing 3.6, DYTAN inserts a self-assignment of variable b if the else branch of the conditional statement, therefore whatever branch is taken the influence of a on b is correctly recognized.

```
int a = 1;  // Taint source - Tainted
int b = 2;  // Taint sink - Not tainted
if (a == 5) {
    b = 3;  // Application of taint propagation logic.
} else {
    b = b;  // Application of taint propagation logic, but
    content value unmodified.
}
```

Listing 3.6. Self-assignment inserted by DYTAN in order to prevent under-tainting.

- A taint sink can take two different content values depending on which branch of a conditional statement with a tainted condition is taken and, while at compile-time the two values appear different, at runtime they are always equal: the source cannot actually influence the sink. In Listing 3.7, variable b takes the value 16 in both branches, so a cannot actually influence it. KLEE-TAINT fails in recognizing this absence of influence and incorrectly determines that variable b is tainted, therefore the analysis suffers from over-tainting. Other tools taken into consideration that enhance basic DTA with static analysis, such as DYTAN and DTA++[7], were not available or usable, but basing on their respective papers it is reasonable to assume that they would behave as KLEE-TAINT.

```
int a = 1;  // Taint source - Tainted
int b = 2;  // Taint sink - Not tainted
if (a == 5) {
    b = sqrt(256);
} else {
    b = pow(4, 2);
}
```

Listing 3.7. Example of over-tainting caused by not taking into account runtime content values.

Given the examples above, it is possible to deduce that a combination of single-path dynamic analysis with static analysis is not sufficient to achieve precise DTA: it is necessary to take into account all the possible runtime values of a taint sink, therefore the analysis must be dynamic and multi-path.

3.4 Multi-path dynamic taint analysis

The approach to DTA proposed in [8] is an application of symbolic execution. The key idea is that taint sources take unconstrained symbolic content values and, for all the execution paths, the content values of the taint sink are observed: if they are constant (i.e. concrete or constant symbolic expressions) and equal across all paths, then the taint sink is not tainted, otherwise, if at least one of the conditions is not met, it is tainted. Unlike in previous traditional approaches, memory locations are not associated with taint values implemented as shadow bits and instructions are not associated with taint propagation logic. The taint status of a sink can only be positive or negative, it is not possible to have a bit array like in KLEE-TAINT.

In Listing 3.8 there are a total of 4 execution paths, as shown in Figure 3.4. At the end of all of them, the taint sink x has the same constant value, thus it is not tainted, the sink y takes only constant values but they are not all equal, thus it is tainted, finally the sink z can take 2 different values at the end of the second path, and being not constant it is tainted. Multi-path DTA, based on symbolic execution, is capable of achieving precise taint "propagation" in such situations.

```
// Environment
void function(int *x, int *y, int *z, int c) {
    if ((c >= 8) && (c < 10)) {
        x = pow(4, 2); y = 3; z = c;
    } else {
        x = pow(2, 4); y = 3; z = 8;
    }
}

// Unit
int main(void) {
    int a, b, c;     // Taint sources
    int x, y, z;     // Taint sinks
    ...
    if (a == 5) {
        if (b == 1) {
            x = sqrt(256); y = 3; z = 8;
        } else {
            function(&x, &y, &z, c);
        }
    } else {
        x = sqrt(sqrt(65536)); y = 6; z = 8;
    }
}
```

Listing 3.8. Example of precise taint "propagation" obtained with Multi-path DTA.

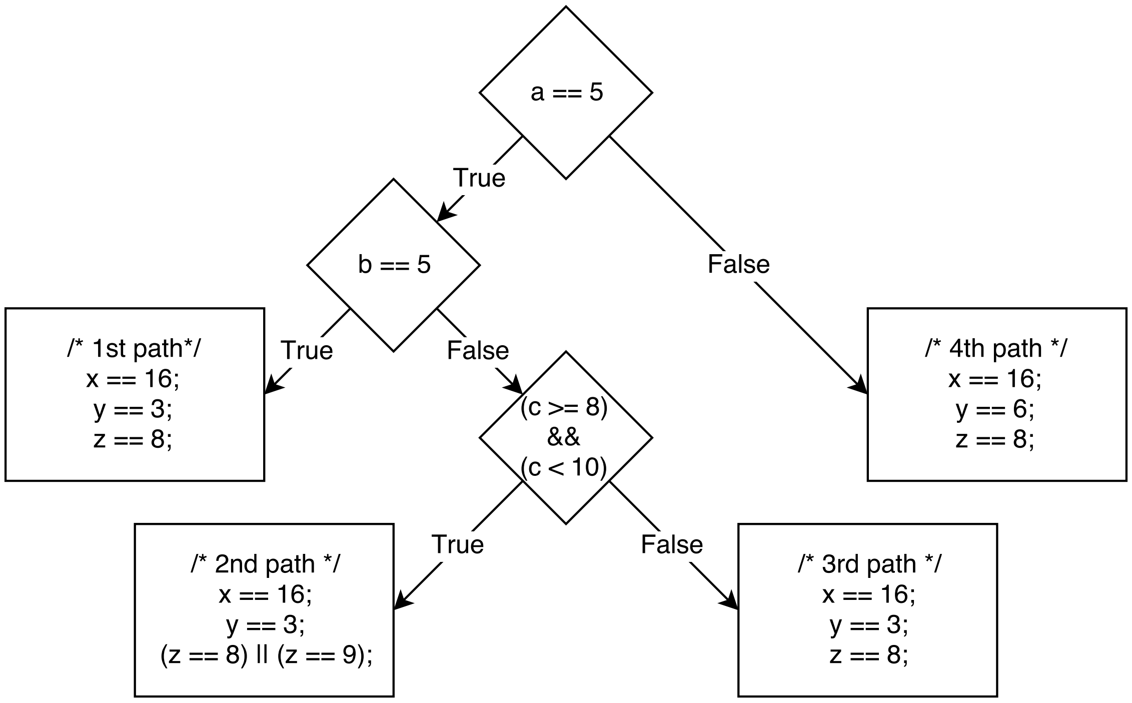

Figure 3.4. Graph of the execution paths of the example in Listing 3.8.

The limitations of multi-path DTA are the same of symbolic execution, specifically the path explosion problem and the solvability of the path constraints. The possible countermeasures depend on its implementation, therefore they will be discussed in Section 5.2.

Chapter 4

Tools for symbolic execution

In order to perform an analysis based on symbolic execution, many state-of-the-art publicly available tools have been taken into consideration, including Angr[9], PySymEmu[10], FuzzBALL[11], KLEE, S2E. As already discussed in Section 2.3, one of the main limitations on symbolic execution is the path explosion problem and, as detailed in Chapter 7, one of the possible countermeasures consists in constraining the symbolic value given as input to the software modules under test. The most straightforward way to enforce these constraints consists in using the so-called source code instrumentation functions provided by KLEE and S2E. These functions allow to inject symbolic values into any variable, not only input ones, and to constrain them by means of standard C language expressions, that are automatically translated into symbolic expressions.

4.1 KLEE

KLEE[4] is a symbolic execution tool that performs execution-generated testing. It requires the C/C++ source code of the program to be tested, which must be compiled to LLVM byte-code in order to be analysed. The output consists in a test-cases for each feasible execution path.

The tool provides two approaches for introducing symbolic values in the program: injection from the environment and intrinsic functions. With the first approach the source code needs no modifications, symbolic values can be injected by means of command line arguments, standard input and files. In Listing 4.1 it is shown that the source code of the program under test is compiled into LLVM byte-code but not linked, then executed with 5 to 10 symbolic arguments of 20 bytes each, with 5 symbolic files and 20 bytes of symbolic standard input available.

```
clang -emit-llvm -c -g program_under_test.c -o
    program_under_test.bc
```

```
klee --sym-args 5 10 20 --sym-files 5 20 --sym-stdin 20
    program_under_test.bc
```

Listing 4.1. Example of injection of symbolic values by means of command line arguments.

The second approach is one of the main innovations in the context of symbolic execution tools: by means of functions provided by KLEE it is possible to inject symbolic values into any variable, at any point of the source code. In addition, other functions allow to arbitrarily modify path constraints, to kill execution paths and to concretize a symbolic expression in order to get sample concrete values that satisfy the constraints at a certain moment of the execution. This approach is used in Listing 4.2: in lines 4 and 5, *klee_make_symbolic()* injects a symbolic value into variable *num*, replacing *scanf()* which asks the user to provide a concrete value; in line 9, *klee_assume()* constrains the set of values that *num* can take, so that the true branch of the if-then-else statement in line 10 can never be executed; in line 15, *klee_get_value_i32()* chooses a concrete value from *num*, so that at most one of the conditional statements in lines 16 and 17 will be executed; finally in line 19, *klee_abort()* terminates each path in which it is executed, so that the function in line 20 is never called.

```
1  int main(void) {
2      int num;
3  
4      //scanf("%d", &num);
5      klee_make_symbolic(&num, sizeof(num), "num");
6  
7      if ((num >= 0) && (num < 10)) {
8  
9          klee_assume((num >= 5) & (num < 8));
10         if (num == 3) printf("A\n");
11         else if (num == 5) printf("B\n");
12         else if (num == 7) printf("C\n");
13     } else {
14 
15         num = klee_get_value_i32(num);
16         if (num == 20) printf("D\n");
17         if (num == 21) printf("E\n");
18 
19         klee_abort();
20         printf("F\n");
21     }
22     return 0;
23 }
```

Listing 4.2. Example of use of KLEE's intrinsic functions.

4.1.1 Architecture

KLEE works as an operating system for processes involving symbolic values and as an interpreter. A symbolic process is a symbolic state, which besides the components of an ordinary concrete process (stack, heap, registers, program counter) includes a path constraint. KLEE directly interprets the LLVM instruction set, uses bit-level accuracy, but lacks support for symbolic floating points instructions, x86 *longjmp* instructions, threads and assembly code (unless the program is recompiled from binary to LLVM instead of from source to LLVM).

Like in pure symbolic execution, state forking occurs in correspondence of conditional statements of which both branches are feasible, but also in correspondence of operations that perform error checking, like divisions; both these types of constraints are treated in the same manner.

Every memory object is represented by means of a distinct STP[12] array, a straightforward representation of the memory as an array of byte is infeasible, because the constraint solver would have to solve too hard constraints. To address the symbolic memory address problem, KLEE performs explicit enumeration: when a pointer can reference N memory objects, the current state is forked N times. To obtain a compact state representation, KLEE implements copy-on-write at memory object level. Allowing common parts of the states to be shared, it becomes possible to test programs that generate a number of states in the order of magnitude of hundred of thousands.

A significant portion of the computational time of an analysis is taken by the constraint solver, thus KLEE implements some optimizations in order to simplify expressions and, when possible, to eliminate queries. Constraint independence divides constraint sets into subsets based on the variables they reference: if the constraint imposed by a branch references variables found only in some on these subsets, its feasibility can be verified with a query that includes only these subsets and the new constraint. In Listing 4.3, before executing the statement in line 5, the path constraint is the conjunction of the conditions of the previous two if-then-else statements; in order to verify the feasibility of the true branch of the third if-then-else statement, its condition must be conjuncted with the path constraint and passed to the constraint solver; it is possible to notice that only the constraints that involve variables a and b are actually needed, the true branch is infeasible regardless of the value of c.

```
1 int a, b, c;
2 ...
3 if ((c < 10) && (b >= 10) && (b < 15)) {
4     if ((c >= 0) && (a == b)) {
5         if (a == 15) {
6             ... // Do something
7         } else {
8             ... // Do something else
```

```
            }
        }
}
```

Listing 4.3. Only the constraints on "a" and "b" are needed to verify the feasibility of the true branch of the third conditional statement.

The Counter-example cache addresses the problem of frequent redundant queries, i.e. queries consisting in a subset or a superset of the constraints of a stored query. If a subset of constraints has no solution, then also the original set has not: in Figure 4.1, the first entry of the cache is a subset of the first query and has no solution, therefore also the query has no solution and needs not to be passed to the constraint solver. If a subset has a solution, it is possible (and often happens) that the solution is a valid also for the the original set (checking a solution against some constraints is cheaper than solving the same constraints): the third entry of the cache is a subset of the third query, its solution allows to simplify the query before passing it to the constraint solver. If a superset of constraints has a solution, then the solution is valid also for the original set: the second entry of the cache is a superset of the second query, its solution satisfies the third query as well.

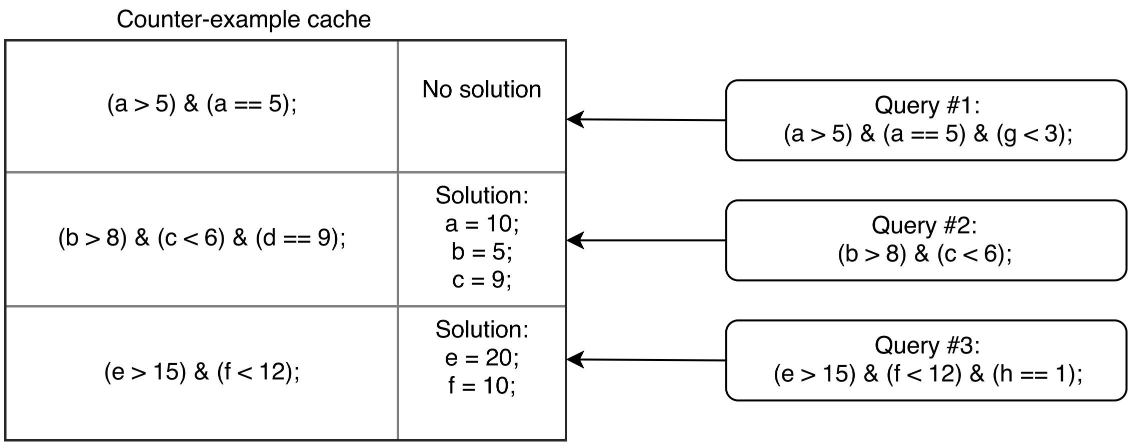
Counter-example cache
(a > 5) & (a == 5);
No solution
(b > 8) & (c < 6) & (d == 9);
Solution: a = 10; b = 5; c = 9;
(e > 15) & (f < 12);
Solution: e = 20; f = 10;
Query #1: (a > 5) & (a == 5) & (g < 3);
Query #2: (b > 8) & (c < 6);
Query #3: (e > 15) & (f < 12) & (h == 1);

Figure 4.1. Example of Counter-example cache.

States are scheduled for execution according to 2 search heuristics, used in a round-robin fashion in order to avoid that one of them gets the execution stuck. Random path selection tends to favour shallow states (that have executed the fewest number of instructions): these states are usually less constrained and thus more likely to explore still unexplored code, and this strategy avoids getting stuck in loops depending on a symbolic condition, which tend to fork lots of states. Coverage-optimized search tries to select states likely to cover still uncovered code.

4.1.2 Environment modelling

KLEE provides a symbolic file system for each execution state, containing a single directory with N symbolic files (specified by a command line argument) and co-existing with the normal concrete file system shared among all states. OS system calls are redirected to models (*open()*, *read()*, *write()*, *close()*, ...) that check if the file to be accessed is stored in the concrete or the symbolic file system. In the former case the corresponding OS system call is invoked, while in the latter case the action is performed on a symbolic file. In Listing 4.4, a simplified version of KLEE's model for the *read()* system call is reported: if the file descriptor provided by the application under test corresponds to a concrete file, then the operating system's *pread()* system call will be invoked, else if it corresponds to a symbolic file, then (part of) a buffer containing symbolic values will be copied into the user-provided buffer.

```
ssize_t read(int fd, void *buf, size_t count) {
    ... // Error checking
    struct klee_fd *f = &fds[fd];
    if (is_concrete_file(f)) {
        int r = pread(f->real_fd, buf, count, f-> off);
        if (r != -1) f->off += r;
        return r;
    } else {
        if (f->off >= f->size) return 0;
        size_t count = min(count, f->size - f->off);
        memcpy(buf, f->file_data + f->off, count);
        f->off += count;
        return count;
    }
}
```

Listing 4.4. KLEE's model for the "read()" system call.

Optionally, also failures of system calls are modelled by specifying a command line option, as shown in Listing 4.5: if the maximum number of allowed failures is set to 1, then when trying to open a symbolic file, the corresponding execution path will be forked into one path in which the file has been opened correctly and one in which the call failed. A successful or failed opening of a concrete file depends on the real file system.

```
klee [...] --max-fail 1 program_under_test.bc
```

Listing 4.5. Failure modelling.

4.2 S2E

S2E[3] (Selective Symbolic Execution) is a tool capable of performing symbolic execution and more in general of analysing the properties and behaviour of software systems by means of multi-path analysis. It is based on the QEMU virtual machine emulator and on the KLEE symbolic execution engine. It features several differences with respect to similar tools: it is capable of full-system analysis and thus to observe programs in their "natural environment"; it can work on unmodified x86, x86-64 and ARM binaries, but it implements also the source code instrumentation approach, with an interface very similar to KLEE, in order allow more flexibility of analysis.

S2E implements the so called selective symbolic execution technique, which allows to restrict the multi-path analysis only the parts of code of interest, while executing all the remaining code in single-path mode (i.e. concretely). This allows to avoid or to reduce the path explosion problem, at the cost of performing a non-fully consistent analysis, but in many cases this is still acceptable. For example, it is possible to test an application without extending the analysis to the kernel code when a system call is made, or to a cryptographic library which gets the analysis stuck because it generates hard to solve constraints.

In order to switch from single-path mode to multi-path mode before calling a given function, it is necessary to substitute the concrete argument values with symbolic values, which can be arbitrarily constrained. In Listing 4.6, the user-provided concrete values are substituted with symbolic values in line 4.

```
 1  char string[3];
 2  string[2] = 0;
 3  //scanf("%2s", string);
 4  s2e_make_symbolic(string, 2, "string");
 5  if (strncmp(string, "OK", 2) == 0) {
 6      printf("OK\n");
 7  } else if (strncmp(string, "NO", 2) == 0) {
 8      printf("NO\n");
 9  } else {
10      printf("Error!\n");
11  }
```

Listing 4.6. Example of switching from single-path mode to multi-path mode.

In order to switch from multi-path mode to single-path mode before calling a given function, it is necessary to concretize all its argument values, i.e. to solve their symbolic expressions and take one of the possible solutions. In Listing 4.7, the analysis is started in multi-path mode, since variable x is assigned a symbolic value in line 2, then, in the execution path that goes trough the true branches of the first two conditional statements, x is concretized and takes value 3, thus only the path where OK is printed is now feasible, the error condition is no more reachable.

```c
int x = 0;
s2e_make_symbolic(&x, sizeof(x), "x");
if (x > 2) {
    if (x < 5) {
        s2e_concretize(&x, sizeof(x));
        if (x == 3) {
            printf("OK\n");
        } else {
            printf("Error!\n");
        }
    } else {
        ...
    }
} else {
    ...
}
```

Listing 4.7. Example of switching from multi-path mode to single-path mode.

To concretize a variable actually means to place an additional equality constraint on it: this is a source of over-constraining, which may prevent the exploration of some otherwise feasible execution paths, making the analysis incomplete.

4.2.1 Execution consistency models

An execution state of a program is consistent if it exists a feasible execution path from the starting state to the current state. Due to the path explosion problem, maintaining consistency during multi-path exploration is often too costly and in many cases it is also unnecessary. S2E offers several execution consistency models, that define trade-offs between the consistency level and the cost of enforcing the model: renouncing to consistency in some parts of the code makes possible feasible and still meaningful analyses. These models rely on the definition of system, unit and environment: the system is the complete software under analysis, the unit is the part of the system to be actually analysed, the environment consists in all the other parts of the system.

- Strictly Consistent Concrete Execution.
  The whole system is executed in single-path mode, the explored path is only determined by the concrete input provided, no symbolic data is involved. The information internal to the system (i.e. the conditions of conditional statement) is not gathered and used. A typical implementation of this model is random input testing, reported in Listing 4.8.

```c
int function(int x) {
    if (x == 4) return 8;    // Explored
    if (x == 6) return 12;   // Explored
    if (x == 8) return 16;   // Not explored
    return x;                // Explored
}
int test_values[5] = {2, 4, 7, 1, 6};
int main(void) {
    int result;
    for (int i = 0; i++; i < 5) {
        if (test_values[i] == 2) result = 4;     // Explored
        else if (test_values[i] == 10) result = 20; // Not
  explored
        else result = function(test_values[i]);
    }
    return (result);
}
```

Listing 4.8. Strictly consistent concrete execution: random input testing.

- Strictly Consistent Unit-level Execution.
  The unit is executed in multi-path mode, while the environment in single-path mode. Only the information internal to the unit is gathered (to form path constraints) and used to explore new execution paths, while paths generated by conditional statements in the environment are missed, causing the exploration of the unit to be potentially incomplete (e.g. this is the case of conditional statement of the unit dependent on values derived from the environment). To implement it, it is necessary to concretize all the symbolic expressions of the data passed to the environment when interfacing with it, as shown in Listing 4.9. KLEE uses this approach when the environment is not modelled, forcing it to concretize all the argument values before executing a system call.

```
int function(int x) {
    if (x == 4) return 8;    // Not explored
    if (x == 6) return 12;   // Explored
    if (x == 8) return 16;   // Not explored
    return x;                // Not explored
}
int main(void) {
    int result;
    int input;
    s2e_make_symbolic(&input, sizeof(input), "input");
    if (input == 2) result = 4; // Explored
    else if (input == 10) result = 20;  // Explored
    else {
        s2e_concretize(&input, sizeof(input));  // Value 6
  is chosen
        result = function(input);
    }
    return (result);
}
```

Listing 4.9. Strictly consistent unit-level execution.

- Strictly Consistent System-level Execution.
  The whole system is executed in multi-path mode, the information internal to both the unit and the environment is used to explore new execution paths, so the analysis is both complete and consistent. Symbolic data is allowed to cross the boundary between the unit and the environment. This model is more expensive to enforce with respect to the previous two, because typically the size of the environment code is orders of magnitude larger than the size of the unit code, so it is more easily affected by the path explosion problem. An example is reported in Listing 4.10

```
int function(int x) {
    if (x == 4) return 8;    // Explored
    if (x == 6) return 12;   // Explored
    if (x == 8) return 16;   // Explored
    return x;                // Explored
}
int main(void) {
    int result;
    int input;
    s2e_make_symbolic(&input, sizeof(input), "input");
    if (input == 2) result = 4; // Explored
    else if (input == 10) result = 20;  // Explored
    else result = function(input);
    return (result);
}
```

Listing 4.10. Strictly consistent system-level execution.

- Local Consistency.
  The unit is executed in multi-path mode, while the environment is not executed or is executed only partially, in single-path or multi-path mode. To implement this model means to substitute (some of) the values returned by the environment with symbolic values that are consistent only with the interface to the environment and that represent any possible valid result of the execution: this allows to explore all the possible paths of the unit, and to avoid executing the environment, obtaining the same performances of the SC-UE model and the completeness of the SC-SE model. The execution state of the unit is always kept consistent, while it is possible to have inconsistencies in the environment, therefore from the point of view of the whole system the analysis is not consistent.

In Listing 4.11, the main function uses another auxiliary function that by contract requires an input value between 0 and 4; if the input is in-bounds, it returns 0 and gives an output value between 0 and 99 according to some external configuration, else it returns -1. The main function asks the user to input a value between 0 and 6, and deals with values 5 and 6 according to some static configuration. It is required to explore all the possible behaviours of the user and to disregard those of the environment, i.e. those of the auxiliary function, therefore the request for user input is substituted with a creation of a symbolic value and the call to the auxiliary function is substituted with a creation of symbolic values for its return value and output parameter, and with the enforcement of its contract's constraints.

```
int mask[5] = {24, 65, 93, 82, 37};
int function(int *out, int in) {
    if ((in >= 0) && (in < 5)) {
        *out = mask[in];
        return 0;
    } else return -1;
}
int main(void) {
    int in, out, res;
    // scanf("%i", in);
    s2e_make_symbolic(&in, sizeof(in), "in");
    if (in == 5) out = 48;
    else if (in == 6) out = 71;
    else {
        // res = function(&out, in);
        s2e_make_symbolic(&out, sizeof(out), "out");
        s2e_make_symbolic(&res, sizeof(res), "res");
        s2e_assume(((res == 0) && (out >= 0) && (out < 100))
    || (res < 0));
        if (res < 0) out = -1;
    }
    return (out);
}
```

Listing 4.11. Local consistency.

- Over-approximate Consistency.
  This model is similar to local consistency, the only difference is that the constraints imposed by the interface of the environment are completely discarded, thus allowing paths of the unit that otherwise would not be explored, because they would normally be infeasible. From the point of view of the system the analysis is not consistent, but it is complete, since every behaviour of the environment is allowed.

With respect to the example in Listing 4.11, the only required modification to enforce this model consists in the removal of the contract's constraints, as shown in Listing 4.12.

```
int main(void) {
    int in, out, res;
    // scanf("%i", in);
    s2e_make_symbolic(&in, sizeof(in), "in");
    if (in == 5) out = 48;
    else if (in == 6) out = 71;
    else {
        // res = function(&out, in);
        s2e_make_symbolic(&out, sizeof(out), "out");
        s2e_make_symbolic(&res, sizeof(res), "res");
        // s2e_assume(((res == 0) && (out >= 0) && (out <
    100)) || (res < 0));
        if (res < 0) out = -1;
    }
    return (out);
}
```

Listing 4.12. Over-approximate consistency.

- CFG Consistency: This model is similar to the SC-SE model in the sense that it exploits the same information to explore new paths, but it is not consistent, since it explores also infeasible execution paths: both branches of every conditional statement are explored, without checking their feasibility with a constraint solver. Every explored path corresponds to some path in the unit's CFG. A use case for this model are dynamic code disassemblers.

4.2.2 Architecture

S2E is composed by a customized version of the QEMU virtual machine, a dynamic binary translator (DBT), and uses the symbolic execution engine of KLEE. For each guest instruction (i.e. of the system under test), the DBT before translation checks if the operands are all concrete: if so, the instruction can be concretely executed by the host CPU and is translated in its instruction set; otherwise the instruction must be executed symbolically by the KLEE's symbolic execution engine and is translated in LLVM byte-code. Avoiding to feed all instructions to the symbolic execution engine, as KLEE does, makes feasible to execute a whole system.

S2E implements a plug-in architecture, in which plug-ins listen for events created by the S2E platform or by other plug-ins. Events are created by the platform when an instruction is translated by the DBT or executed by the virtual machines, when memory is accessed, when an interrupt is issued, when a state forks, etc... When an execution path creates an event the listening plug-ins are able to access and modify the corresponding state, which includes physical memory, registers, program counter, PID of the current process, etc... Plug-ins are able to access also the global state.

In order to instrument the source code of a program to be tested, S2E provides a header file containing functions for creating symbolic values, concretizing symbolic expressions, printing debug information, etc... These functions, that are meant to be executed by the guest system, use custom op-codes which S2E interprets and for which provides the actual implementation.

To solve the symbolic memory address problem, S2E identifies all the memory pages that can possibly be referenced and passes all of them to the constraint solver. In order for this solution to be efficient, the page size must be small, in the order of hundreds of bytes; S2E allow the user to configure it.

Chapter 5

Tools for dynamic taint analysis

As detailed in further chapters, a part of this work consisted in testing exhaustively the compliance of a message gateway with its specifications, by means of dynamic taint analysis. This requires precise taint propagation/deduction, thus the tools used have been selected according to their capability to provide it. The TEMU[13] and Taintgrind[14] tools propagate data-flow taints but not the control-flow taint, thus they have been discarded. The DYTAN, DTA++ and KLEE-TAINT tools enhance the dynamic part of the analysis with static analysis in order to propagate the control-flow taint. Among these, only the last one was publicly available and used. However, only multi-path DTA is virtually capable of precise taint deduction and, since there were no publicly available tools using this approach, it has been implemented on top of S2E.

5.1 KLEE-TAINT

KLEE-TAINT[5] is an extension of the KLEE tool that adds the capability of performing Dynamic Taint Analysis. It extends the LLVM byte-code interpreter in order to implement in a straightforward manner the taint propagation logic. Taints are arrays of bits, combined (merged) by means of the bitwise OR operator. To the best of the author's knowledge, this is the only tool for dynamic taint analysis that implement the source code instrumentation approach: it provides 2 functions, *klee_get_taint()* and *klee_set_taint()*, that respectively mark as tainted and check the taint value of a variable. The tool allows to choose among 3 taint propagation modes: direct (data-flow tainting), indirect (data-flow and control-flow tainting) and region-based (precise data-flow and control-flow tainting).

- In direct flow mode, for each operation the taints of all the operands are merged together in order to obtain the taint value of the result. If an assignment to a variable depends on a tainted condition, the taint value of the variable does not take into account the taint value of the condition: this means that this mode introduces under-tainting. In Listing 5.1, when the content values of variables a and b are summed, their taint values are merged into value 3.

```
int a = 1;
int b = 100;
int c = 0;
klee_set_taint(1, &a, sizeof(a));
klee_set_taint(2, &b, sizeof(b));
c = a + b;
int taint_c = klee_get_taint(&c, sizeof(c));
```

Listing 5.1. Direct flow mode example.

- In indirect flow mode, when the condition of a conditional statement is dependent on a tainted value, then the control flow is tainted by assigning to the program counter the merging of the taint values of the involved variables. All subsequent assignments to variables are merged with this taint value. Once the PC is tainted, it cannot be untainted even if two branches converge to common instructions, that are thus executed independently of the condition. This means that this mode introduces over-tainting, but it can still be useful in cases when a tainted PC is an error by itself, for example in cryptographic routines where data undergoing some elaboration must not be able to determine the control flow. In Listing 5.2, the program counter takes the taint value of variable a, which is then propagated to variable c. Therefore, the taint value of c will be 1.

```
int a = 1;
int c = 0;
klee_set_taint(1, &a, sizeof(a));
if (a == 1) {
    c = 1;
} else {
    c = 0;
}
int taint_c = klee_get_taint(&c, sizeof(c));
```

Listing 5.2. Indirect flow mode example.

- Region-based mode aims to achieve precise taint propagation: in order to do so, when two branches of a conditional statement converge to instructions that are executed independently of the condition, the control flow (i.e. the program counter) is un-tainted. Specifically, the program counter is associated with a stack of taints: when executing a conditional statement the current PC taint is pushed into the stack and merged with the taint of the condition, in order to obtain a new PC taint to be used for all the assignments inside the two branches; after exiting the selected branch the previous PC taint is popped and restored as the current PC taint. In this way, variables assigned independently of the condition are not over-tainted. In Listing 5.3, before checking the condition in line 5, the current taint value of the PC, 0, is pushed into a stack, then it is updated to the taint value of a, which is 1. This value is propagated to variable b and then, when exiting the branch, it is discarded and the previous value is pulled from the stack, therefore c will have taint value 0.

```
1  int a = 1;
2  int b = 0;
3  int c = 0;
4  klee_set_taint(1, &a, sizeof(a));
5  if (a == 1) {
6      b = 2;
7  }
8  c = 3;
9  int taint_b = klee_get_taint(&b, sizeof(b));
10 int taint_c = klee_get_taint(&c, sizeof(c));
```

Listing 5.3. Region-based mode example.

Before performing an analysis in region-based mode, KLEE-TAINT performs a static analysis on the LLVM byte-code of the program under test in order to identify its SESE (single-entry single-exit) regions[15], which are sequences of instructions terminating with an instruction that alters the control flow and that have a single entry point and a single exit point. These regions are connected together in a control flow graph, and each of them is assigned a level, which is used to perform push and pull operations on the stack of the PC's taints: when the level increases by one, the current taint is pushed into the stack, when it decreases by one, the previous taint is pulled from the stack. A portion of C code consisting in if-then-else statement preceded and followed by other instructions, like the one in Listing 5.4, corresponds to at least 4 SESE regions connected to form a control flow graph like the one in Figure 5.1 (for simplicity the C instructions are reported instead of the LLVM ones): the instructions preceding the body of the conditional statement and its condition are inside one region, the two blocks delimited by curly braces are inside one region each of a superior level, and finally the instructions following the body are inside a region of the same level as the first one. When taking one of the branches of the conditional statement, the level increases thus the current PC taint value is saved, when exiting one of the branches, it decreases thus the previous PC taint is used and the taint of variables e and f is determined correctly.

```c
int a, b, c, d, e, f;
...
b = 2 * a;
if (c == 5) {
    c = 6;
    d = c + b;
} else {
    c = 7;
    d = c - b;
}
e = a + d;
f = c + e;
...
```

Listing 5.4. Example of C code including an if-then-else statement.

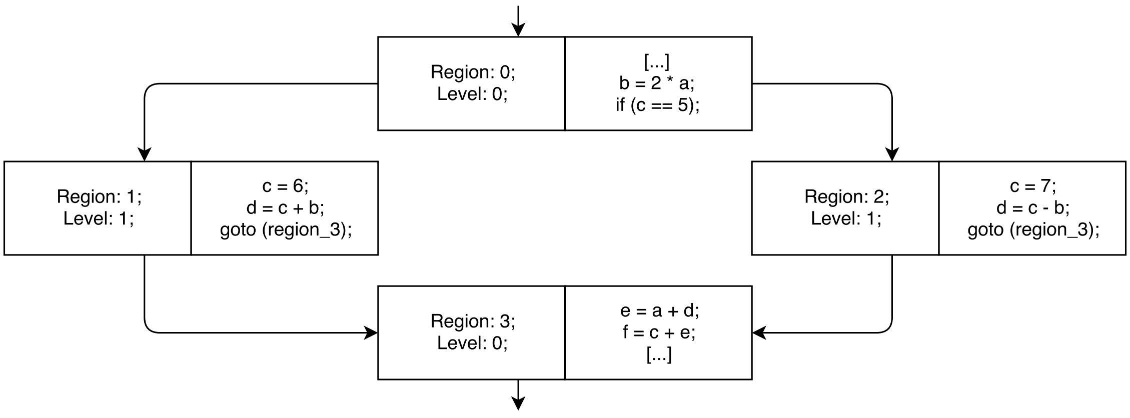

Figure 5.1. Portion of the control flow graph corresponding to the code in Listing 5.4.

5.1.1 Limitations

Even when performing an analysis in region-based mode, KLEE-TAINT is not capable of detecting all the information flows, as shown in Listing 5.5: the user is asked to input a single character which is marked as tainted and used to index an array; the indexed cell is set to 1, while all the other cells retain value 0; a for loop is used to increment the value of variable b until it is equal to the value of a; the taint value of b should be equal to the one of a, but it remains 0. The information flow between a and b is indirect, because is not caused by a simple assignment, and goes undetected, thus this is an example of under-tainting.

```c
int v[256] = {0};
```

```
unsigned char a, b;
scanf("%c", &a);
klee_set_taint(1, &a, sizeof(a));
v[a] = 1;
for (b = 0; v[b] != 0; b++);
int taint_b = klee_get_taint(&b, sizeof(b));
```

Listing 5.5. Example of information flow not detected by KLEE-TAINT.

KLEE-TAINT is incapable of detecting also other types of information flows, which results in over-tainting, but this is a problem that affects also other similar tools and has a common cause in all of them. Section 3.4 details this problem and proposes a possible solution.

5.2 Implementation of multi-path DTA

The implementation of the algorithm for multi-path DTA proposed in [8] is not publicly available. A new prototype has been developed on top of S2E as a plug-in.

On the guest side, two instrumentation functions, reported in Listing 5.6, were added to the S2E header file: they allow to declare a generic array (or a variable, as a particular case) respectively a taint source or a taint sink.

```
static inline void s2e_taint_source(void* buffer, size_t size,
    const char *name);
static inline void s2e_taint_sink(void* buffer, size_t size,
    const char *name);
```

Listing 5.6. S2E instrumentation functions for Multi-path DTA.

Sources and sinks need to be named: for the former, the only purpose is to be able to print more detailed debug information; for the latter, names are necessary to distinguish them, as explained in detail later. Taint values cannot be manipulated directly like with the instrumentation functions provided by KLEE-TAINT for two reasons. There are only two possible values, declaring an array as a source means to assign to it the positive taint value. S2E provides the user with a debug console and with a window for the guest's standard output, which is shared among the various execution paths and reset as soon as the corresponding path terminates. Therefore it is convenient to print all the analysis data on the debug console rather than on the standard outputs, making useless to have the taint values of the sinks inside the guest.

On the host side, the implementation of the plug-in consists in a C++ class that maintains a global state over all the execution states. This state consists in a map with the correspondence between an identifier and a mapped value for each taint sink: the former is a conjunction of the address, the size and the name of the buffer used as sink, while the mapped value is a structure containing a flag stating if the sink is tainted or not and a dynamically allocated vector of bytes containing a copy of the initial value of the taint buffer. The class provides a public function, *onCustomInstruction()*, registered as a callback to be called each time an assembly instruction with a custom op-code of the guest is executed. This function intercepts two custom op-codes, one for each instrumentation function available to the guest, and provides their actual implementations. The code for *s2e_taint_source()* is fundamentally the same as *s2e_make_symbolic()*: it creates a symbolic buffer of the required size and copies it in the specified memory cells. The only difference consists in the more detailed information printed on the debug console. The code for *s2e_taint_sink()* distinguishes among several cases depending on the provided sink identifier. If the identifier is not found inside the map, it means that a new taint sink has been declared and it is memorized. Then its initial content value is examined: if it is constant (i.e. a concrete value or a constant symbolic expression), then it is memorized together with the key and it is deduced that the sink is not tainted so far, otherwise the taint status flag of the mapped flag is raised. If the identifier is found inside the map and the taint status flag is low, then the current content value is examined: it must be a constant and it must be equal to the initial value; if one of these condition is not met, then the sink is tainted and the taint status flag is raised. Finally, if the identifier is found and the taint status flag is high, there are no operations to be performed, if a sink has already been proved tainted, it cannot be untainted. Once the possibility that a taint can be propagated to a sink has been proved, the analysis can be stopped, while to demonstrate that it cannot propagate, it is necessary to run it to completion. The implementation of this prototype is straightforward, as it only required slightly less than 300 lines of codes, including comments. Nonetheless, its effectiveness has been successfully verified on the test suite of KLEE-TAINT (which is the most advanced tool that was publicly available due to its capability of propagating the control-flow taint) and then on all the critical examples described before where the traditional approach to DTA fails.

5.2.1 Execution consistency models

Since the proposed implementation of multi-path DTA is based on S2E, in order to limit the intrinsic problems of symbolic execution it is possible to enforce one of the consistency models, trading the feasibility of the analysis for the possibility of incurring in over-tainting and/or under-tainting. To describe these effects, the example in Listing 3.8 is considered again: the main function is part of the unit and the auxiliary function is part of the environment; out of four paths, two pass through the environment.

- If the SC-SE model is applied, then the analysis is precise. If the execution engine is not capable of completing the analysis (i.e. to run all the execution paths past the taint sink), due for example to loops or recursions that depend on symbolic conditions and that must be quitted without fully exploring them, there can be under-tainting, because not all the possible values of the taint sink are taken into consideration. For instance, if the fourth path in Figure 3.4 is not fully explored due to early termination, it is not detected that the sink y takes value 6, and considering only the other 3 paths where it takes always value 3, it is wrongly deemed not tainted.

- If the SC-UE model is applied, there may be under-tainting: concretizing parameter values before environment calls causes the loss of feasible paths, and therefore of possible values of the taint sink. For instance, if the input parameter c is concretized to value 5 before calling the environment function, then the second path in Figure 3.4 is not explored, and the sink z is wrongly deemed not tainted.

- If the LC and the OC models are applied, there may be over-tainting: substituting values returned by environment calls with under-constrained or unconstrained symbolic values allows the taint sink to take values that otherwise would be impossible. For instance, if the environment function is not called and its output parameters x, y and z are substituted with unconstrained symbolic values, then the sink x will be erroneously considered tainted.

Chapter 6

Use case

The use case selected for the testing methodologies described before is a central messages gateway residing inside one of the ECUs of a car. CAN buses are used to interconnect all the ECUs of the car and to interface with external diagnostic devices; there are a total of 6 CAN buses, all of them are connected to the central gateway ECU, some are reserved for the communication among the internal ECUs and some for diagnostics, as shown in Figure 6.1.

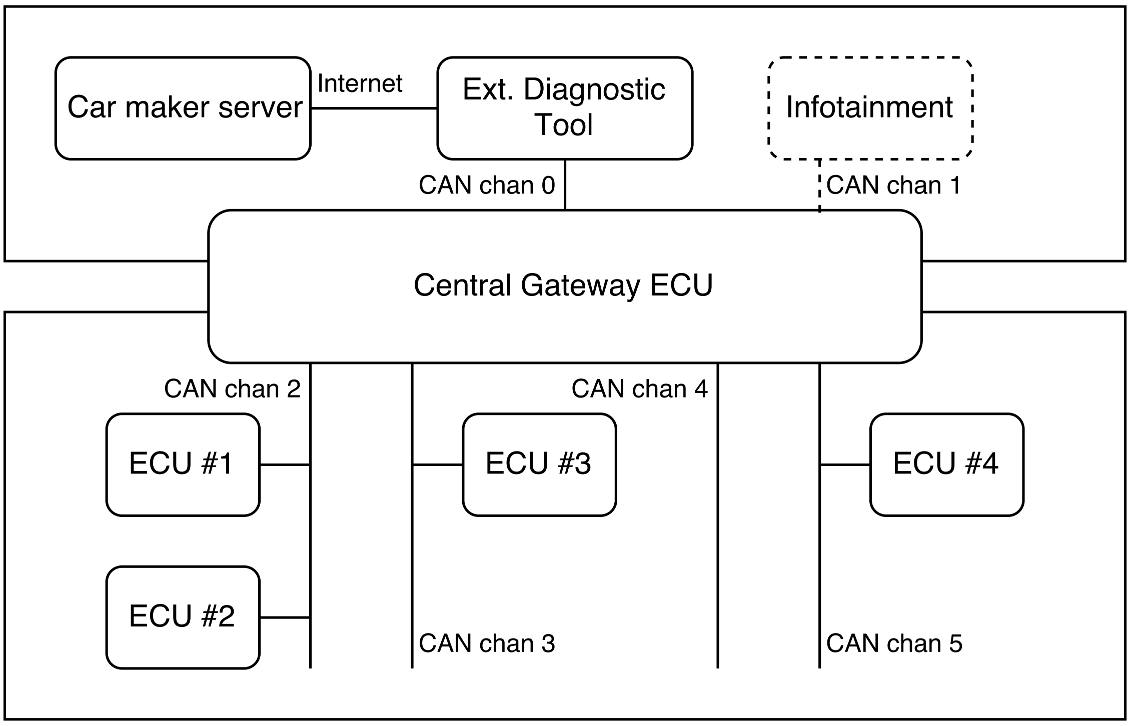

Figure 6.1. Interconnections between the central gateway ECU and the others ECUs.

The central gateway ECU provides two main functionalities: filtering and routing of CAN messages coming from and directed to internal ECUs depending on their compliance with the specifications and their frequency of exchange, and allowing the exchange of messages from and to external diagnostic devices only if they complete an initial authentication phase. CAN messages are not encrypted nor authenticated, cryptographic authentication is only used to verify the identity of external diagnostic devices.

The gateway ECU features a quad core ARM-based processor, three of which are general purpose cores and one is a HSM, a hardware-accelerated cryptographic co-processor. Currently only one of the general purpose cores is used. The ECU runs a real-time operating system, which performs scheduling of the tasks and provides a hardware abstraction layer for interfacing with the CAN bus modules and with the HSM core. The low-level interface between the host (the 3 general purpose cores) and the HSM is implemented with RAM buffers and interrupts, as shown in Figure 6.2: when an application task running on the host needs the HSM to execute a cryptographic computation, it writes the necessary data on a RAM input buffer, then it issues an HSM service request, which triggers an interrupt service routine in the HSM, and then prosecutes with other computations, since the request in non-blocking; the HSM reads the input data, executes the code of the selected interrupt service routine, and writes back the output data in another RAM output buffer; the application task on the host can verify the progress of the operations on the HSM with a polling-based interface and it retrieves the results once they are completed.

The firmware running on the ECU is organized into several modules whose interconnections and interactions are illustrated in Figure 6.3. When a CAN message is injected in one of the input channels, the CAN Interface module determines if it is directed to an internal ECU (normal operation or diagnostic message) or it is an authentication message. In the former case, it is redirected to the Communication module that invokes, depending on the message identifier, a callback of the Content-based Message Filter and one of the Frequency Monitor; if the message must be forwarded, the CAN Interface sends it back to the driver. In the latter case, it is redirected to the CAN Transport Protocol and to the Diagnostic Protocol that delegates certificates validation and cryptographic computations to the Certificate Manager.

6.1 CAN messages filtering and routing

6.1.1 Modules

The CAN Interface is the lowest level module: it is used both by the modules that perform message filtering and by the modules that perform authenticated diagnostic access. It is statically configured with a list that, for each allowed incoming message

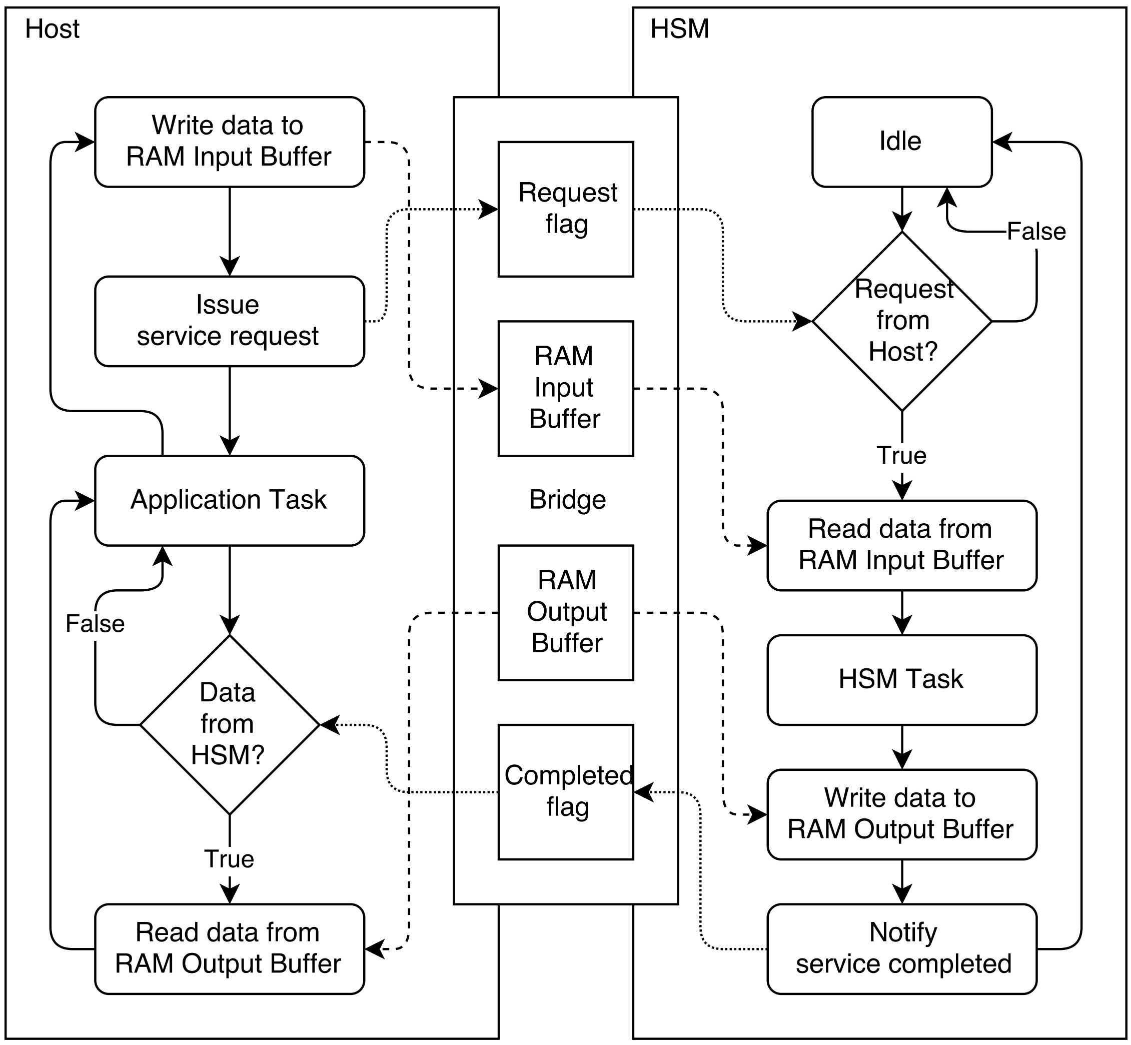

Figure 6.2. Interface between the host and the HSM.

type, includes the message identifier, the length code, from which CAN channel(s) it is expected to be received, to which CAN channel(s) must be sent and which upper level modules must read the payload before an eventual forwarding (message filtering modules) or to perform authentication (authenticated diagnostic access modules). This module therefore provides low level gateway functionalities, being able to route packets, to filter out packets with an unknown identifier, with a wrong size and/or coming from a wrong channel; it does not inspect in any way the payload of the messages.

The Communication Module interfaces with the CAN Interface, with the Content-based Message Filter and with the Frequency Monitor. It receives CAN messages

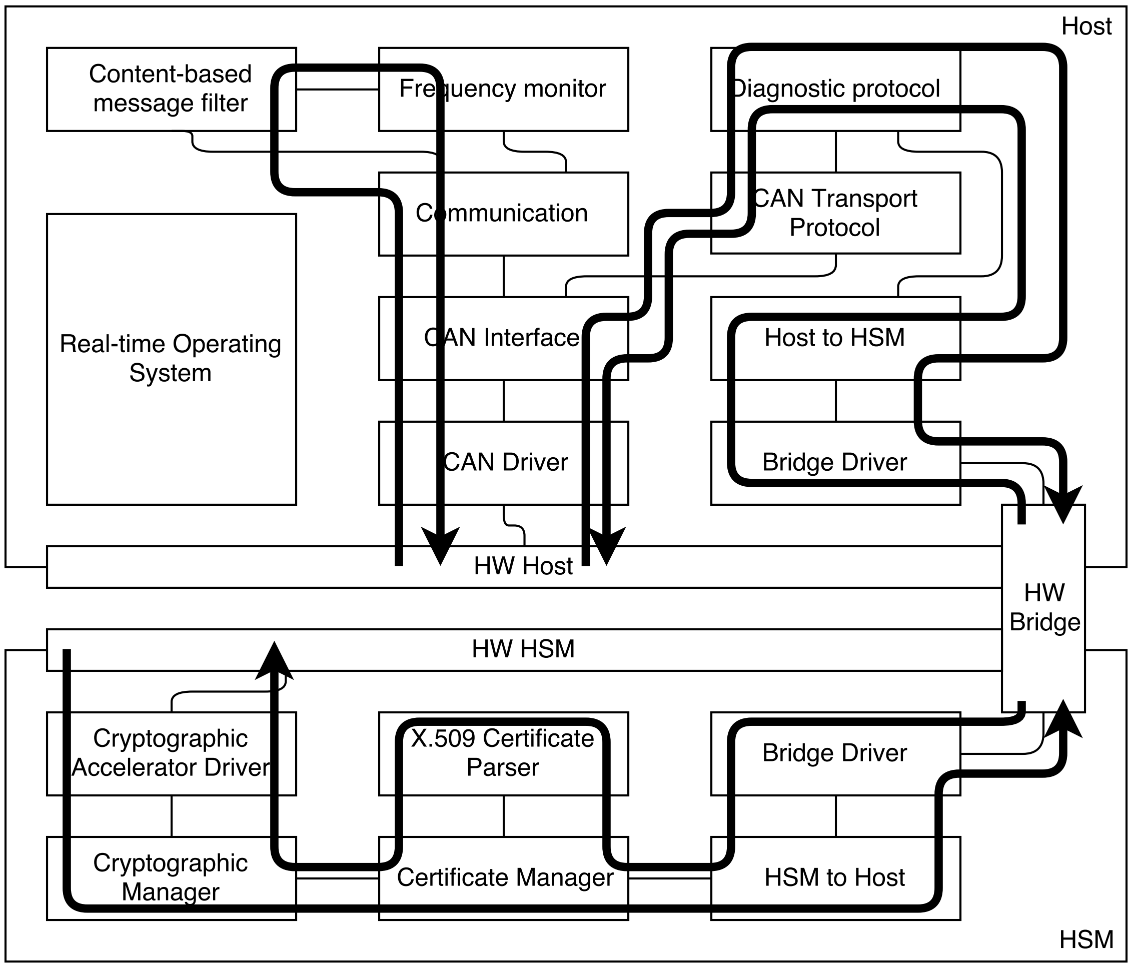

Figure 6.3. Software modules and interactions of the ECU firmware.

from the CAN Interface and, according to their identifier, invokes the corresponding callbacks with the appropriate parameters of the Content-based Message Filter and/or the Frequency Monitor.

The Content-based Message Filter is a module that inspects the payload of the messages and performs a white-list/black-list check. First, it checks if the incoming message is of diagnostic type, in which case it lets the message pass, regardless of its content, only if an authenticated diagnostic access was previously completed successfully. If the message is of any other type, then its payload the signals in the payload are isolated and checked individually. Each signal value must be inside a specific range, which is specified statically in the module configuration, otherwise the whole message is blocked.

The Frequency Monitor is a module that checks how frequently a message directed to other ECUs tries to pass through the gateway ECU. If that type of message is configured to have an upper bound on the frequency, and if this frequency is exceeded, the message is blocked and a procedure to log the anomaly is invoked.

6.2 External diagnostic devices authentication

Authentication of external diagnostic devices is divided in two phases: the first one consists in the verification, by the gateway ECU, of the digital signature of a X.509 certificate provided by the external device; the second one consists in a challenge, made by the gateway ECU, against the external device, which has to prove the knowledge of the private key corresponding to the public key of the certificate. As already mentioned, communication is done over CAN buses, whose frame payload is limited to a maximum length of 8 bytes. A typical X.509 certificate is about 1000 bytes long, therefore a transport layer is necessary. The adopted standard for the transport layer is ISO-TP[16], whose segment payload can be up to 4095 bytes long. ISO-TP can use its own addressing in conjunction with the CAN identifier (Extended Addressing), reserving the first byte of the frame payload for it, but this operation mode is not used (Normal Addressing). As shown in Table 6.1, ISO-TP defines four frame types, each identified by a code in the 4 most significant bits of the first byte: single frame, whose payload contains the whole segment payload (which can be up to 6 or 7 bytes long) and the segment payload length in the 4 least significant bits of the first byte; first frame, whose payload contains the first part of the segment payload (that has a minimum of 7 or 8 bytes and a maximum of 4095 bytes) and the segment payload length in 12 bits, starting from the 4 least significant bits of the first byte and including the whole second byte; consecutive frame, whose payload contains subsequent data of the segment payload and the index of the segment in the 4 least significant bits of the first byte, which is thus reset every 16 segments; control flow frame, sent by the receiver to acknowledge a first frame, containing parameters for the transmission of further segments.

6.2.1 Modules

The CAN Transport Protocol module implements the ISO-TP standard for the transport layer. It interfaces with the CAN Interface to send and receive CAN messages, and with the Diagnostic Protocol, which uses it to execute the two-phase authentication procedure.

The Diagnostic Protocol module implements the two-phases authentication procedure. It features two main procedures, each one corresponding to one of the two phases, that are triggered by a specific message received by the CAN Transport Protocol: the first one requires the HSM to verify the signature of a given X.509

	Byte 0: bits 7..4	**Byte 0: bits 3..0**	**Byte 1**	**Byte 2**	**...**
Single frame	0	Size (0..7)	[Data 1]	[Data 2]	[Data 3]
First frame	1	Size (8..4095)		Data 1	Data 2
Consecutive frame	2	Index (0..15)	[Data 1]	[Data 2]	[Data 3]
Flow control frame	3	Flow control flag	Block size	Separation time	

Table 6.1. ISO-TP frame types and content.

certificate and to provide a cryptographic random number to be used as a challenge with the provider of the certificate, which has to prove the possession of the corresponding private key; the second one verifies the result of the provided challenge and, if it is correct, it grants the forwarding of diagnostic messages. These procedures are executed relying on the Host to HSM module.

The Host to HSM module and the HSM to Host module are two complementary modules, the former runs on the host, the latter on the HSM. They both are interfaces to the operating system's driver for the HSM bridge: the former provides functions to verify a X.509 certificate, to generate a cryptographic random number, to verify the the result of a cryptographic challenge and to verify the status of these requests; the latter listens to service requests and when one is issued it invokes the corresponding function of the Certificate Manager module passing the received parameters.

The Certificate Manager module is an application of the HSM core that implements the high level procedures for verification of X.509 certificates and verification of the result of cryptographic challenges. To perform these tasks it relies on two modules: the Cryptographic Manager and the X.509 Certificate Parser; the former is a simple wrapper for the operating system's driver for the cryptographic hardware accelerator, it allows to verify the content of a certificate given the signature and the RSA public key and to perform basic encryption and decryption of byte arrays given a RSA public key. This module is statically configured with the certificates of the authorities that emit allowed certificates. An incoming X.509 certificate is valid if its binary encoding is correct, if its issuer is equivalent to the subject of one of the statically configured certificates, if the signature is valid and if it is not expired. The challenge-response authentication consists in sending a cryptographic random number to the external device, who encrypts it with the private key of the issuer of the incoming X.509 certificate and sends back the result, and in decrypting this result with the corresponding public key; if the result of the decryption is equal to the number sent, then the external device is authenticated.

6.2.2 X.509 certificates parsing

Both the statically configured certificates and the incoming certificates are encoded following the ITU-T X.690[17] standard, specifically according to the ASN.1 Distinguished Encoding Rules (DER) format. DER is a self-describing, self-limiting TLV (type, length, value) format: each encoded element consists in identifier octets (i.e. bytes), length octets and content octets, as shown in Table 6.2.

Identifier octets (1+)	Length octets (1+)	Value octets (0+)	End-of-content (not allowed in DER format)

Table 6.2. High view of the X.690 encoding structure.

The first identifier octet consists in 2 bits for the tag class, that specify if the tag number is an ASN.1 native type or dependant on the application or on the context, 1 bit that states if the content encodes directly the element value (primitive) or contains more element encodings (constructed), 5 bits for the tag number, that identifies an ASN.1 native type (e.g. boolean, integer, floating point number, bit string, character string, set, sequence, ...) or another custom type. If the tag number is 31, the real tag number is encoded in further octets because it cannot fit in 5 bits (long form): in this case the following octets consist in 1 bit that states if there are further identifier octets and 7 bits for (a part of) the tag number.

First octet								**Second octet**								**Nth octet**							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Tag class		Prim. / Constr.	Tag number (0..30)					N/A								N/A							
			Long form (31)					More id octets	Tag number							More id octets	Tag number						

Table 6.3. Identifier encoding structure.

The first length octet consists in 1 bit that states if the length is in short form or in long form: in the former case, the subsequent 7 bits define the number of octets of the content (0 to 127); in the latter case, the subsequent 7 bits define the number of further length octets (1 to 126, 127 is a reserved value, 0 corresponds to the indefinite form, which is not allowed for DER), that contain the actual content's length.

First octet								**Second octet**								**Nth octet**							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Def. short (0)	Num. value octets (0..127)							N/A								N/A							
Def. long (1)	Num. length octets (1..126)							Num. value octets															

Table 6.4. Length encoding structure.

DER encoding of ASN.1 values is unambiguous: when a given data structure is serialized, the obtained representation is unique. This feature is required when a data structure needs to be digitally signed, which is the case of X.509 certificates, for which DER is suitable and widely used. To achieve uniqueness of the representation, DER prescribes to use the shortest possible length encoding, to use the primitive encoding for bit strings, octet strings and restricted character strings, and to sort the elements of a set according to the tag number.

Parsing a X.509 DER-encoded certificate is an operation that can theoretically be subdivided in two distinct phases: generic de-serialization with syntax checking and verification that the de-serialized structure corresponds to a certificate with all the required fields. The X.509 Certificate Parser module intermixes these two phases: for each certificate field, it decodes a portion of the encoded binary and verifies that it corresponds to the expected field. This procedure, as described in Chapter 9, is error prone but, in this context, it is the easiest approach. Decoding and copying a generic DER-encoded ASN-1 type requires dynamic memory allocation, because it is impossible to know a priori the length of the content, but the operating system of this platform does not offer any support for it, thus only static allocation is possible. It is not possible to dynamically construct a structure where the presence and the size of fields corresponding to ASN-1 types depend on the parsed binary, it is only possible to statically allocate a structure with the required fields of a X.509 certificate, that contains pointers to the binary. From this constraint comes the necessity to intermix the two phases.

Chapter 7

Template procedure of the analysis

Before reporting the results of the analysis on the central gateway software, the template procedure for an analysis based on symbolic execution, which includes multi-path dynamic taint analysis, will be discussed.

Construction of the test harness

The test harness is made up of three main components, as depicted in Figure 7.1: the software modules to be tested, which are unmodified with respect to the original system, the main testing functions and the stub modules, which are optional.

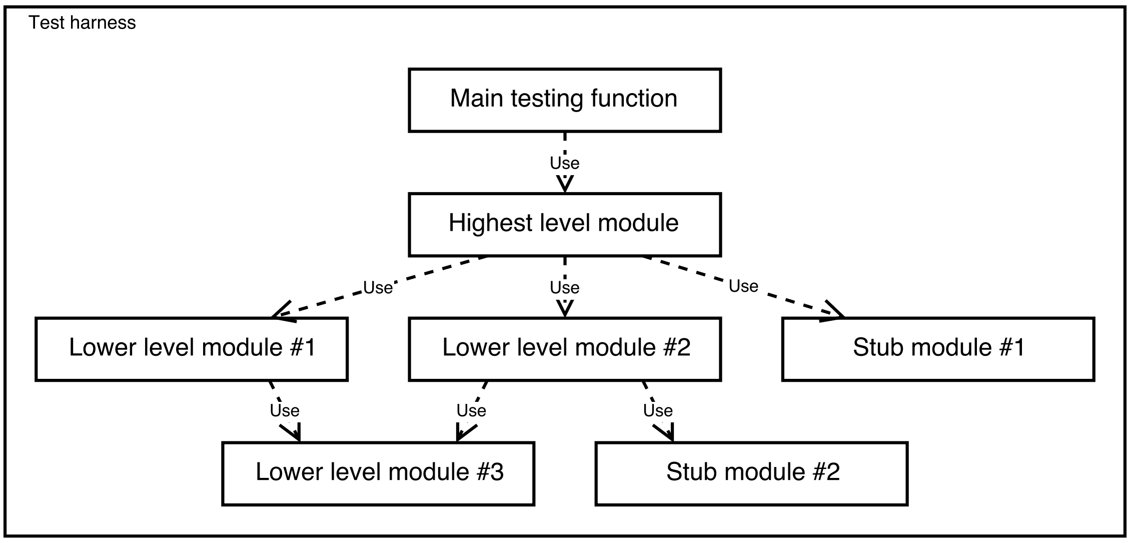

Figure 7.1. Structure of a test harness.

In general, a software module provides a public interface composed by function signatures and data structures, and uses the public interfaces of other modules. The main function uses only the interface of the highest-level module, which is the one on which the analysis is focused. The highest-level module may need the functionalities offered by lower-level modules, which must either be included into the test harness or substituted with stubs that implement their interface. The reasons that lead to choose the former or the latter solution will be clarified later.

Main function of the test harness

The testing of a module can be done function-by-function or according to specific procedures, depending on its design. The former approach is suitable for re-entrant code, the latter when the code relies on global and/or static variables. For example, if an interface offers a function to pass data to be elaborated and another function that performs the actual elaboration, testing these functions singularly would have little to no meaning, therefore the main testing function should call them in the intended order. In the first case, the user-controlled inputs coincide with the input parameters of the functions; in the second case, they are a subset of the signature parameters, and must be identified according to the semantic of the procedure. The UC-KLEE[18] tool (under-constrained KLEE) offers support for function-by-function testing of modules with interfaces intended to be used according to procedures, by implementing the so-called lazy initialization. When a symbolic input pointer is dereferenced for the first time, the execution forks: in one path, it is given a null value; in the other one, a new object of its type is allocated and bound to it. The analysis of a function with this approach is under-constrained, as the preconditions on the inputs are not enforced, thus it yields false positives, but it is capable of exploring otherwise unreachable code due to path explosion. UC-KLEE has not been used here since it was not publicly available.

Inputs and outputs

Before calling functions of the module under test, part of or all the user-controlled inputs must be made symbolic. This choice mainly depends on the feasibility of the analysis due to the path explosion problem: if the module's logic is prone to it, a possible approach is to maintain some of the inputs concrete or to constrain them, in order to try to explore only one section at a time. This possibility will be detailed later.
If requested for the kind of analysis performed, the outputs of a function/procedure can be concretized and observed inside the test harness. It is possible to do the same also for the inputs, but it is necessary to do it after the execution of a function/procedure and after the concretization of the outputs. This is due to the inner working of the KLEE tool (and of the S2E tool, which relies on it): concretizing a value stops path forking (a concrete value can trigger the execution of one and

only one branch of a conditional statement) and thus the exploration of all possible behaviours (execution paths) of the code. KLEE does this automatically anyway to generate test cases for the inputs that were made symbolic, but often the mixed hexadecimal/ASCII printing format makes hard to interpret them manually.

Stubs

When a lower-level module causes unmanageable path explosion, too hard to solve constraints and/or it relies on unavailable hardware, it is possible to substitute it with a stub that implements its interface. A typical example of modules that need to be stubbed so that the analysis yields meaningful results in a feasible time are those that perform cryptographic computations. These stubs make no use of the input parameters, while part or all of the output parameters and the return value are made symbolic. These value are conceptually equivalent to the user-controlled inputs in the main testing function, therefore they can be concretized and observed inside the the test harness, with the same limitations described above. A stub returns completely unconstrained values, it does not apply any of the constraints derived from the logic of the modules they substitute, therefore the whole analysis is under-constrained and may yield false negatives. When a supposed bug-triggering concrete input value is found, it is necessary to verify that it actually is by reinserting the original module in place of its stub and running the function/procedure under test with it.

Test cases replay

Once a test case has been generated, it is possible to replay it either with the KLEE-REPLAY tool or by embedding it into the test harness. In the former case, the test harness is compiled in LLVM and interpreted in the same environment as KLEE. In the latter case, it is compiled in assembly and executed natively; this requires to disable the instrumentation functions calls provided by KLEE and usable only inside its environment, specifically to substitute all the function calls that return symbolic values (i.e. *klee_make_symbolic()*, *klee_range()*, ...) with the corresponding section of the test case, both in the main testing function and in the stubs, if present. In case the test case triggers a bug, the former approach is suitable for evidencing the "logical" aspect of the error, while the latter approach allows to observe the effects of the error on the real platform. For example, a memory error caused by a sequence of out-of-bounds accesses to an array is detected by KLEE at the first wrong access and this behaviour is replicable with KLEE-REPLAY; the same error may not be detected when the same source code is compiled in assembly and executed in the real platform, or not immediately but only when the Nth wrong access causes a segmentation fault or an exception.

Path explosion taming

As mentioned before, making all the user-controlled inputs symbolic and unconstrained is often a too naive approach, as it may easily lead to path explosion. If the function/procedure expects as input an array with its relative size, it is possible to address the problem by constraining the size. A typical situation that occurs with variable sized arrays is a loop iterating on each element, which means that for each integer value that the index can take (for example, from 0 to size minus one) a new execution path is forked; the smaller the element and the larger the size, the more paths are created. Furthermore, if the size is completely unconstrained, it mismatches the actual size leading to false positive errors: the called has to trust the caller for consistency between the size passed as parameter and the actual one. By leaving the size concrete, this source of path forking and false positives is removed and the analysis focuses on the content of the array. Making the size symbolic and then constraining it so that it can take a range of values between 0 and the actual size, allows to avoid the false positives and to reduce the path explosion. Another possibility, that does not exclude the previous one, consists in constraining the content of the array: it is useful and often necessary when the input undergoes elaborations, like decoding and parsing, in which the control flow is heavily dependent on user-controlled data, and that thus are prone to path explosion. For example, if the array contains the binary encoding of a structure, it is possible to make symbolic only part of the fields: this limits path forking and thus the extension of the exploration, but it allows to reach quicker deeper paths, i.e. to explore functions that parse only a particular field or all fields of a particular type, triggering deep errors otherwise unreachable in a feasible time.

Influence of the inputs on the control flow

An important result of an analysis is the number of generated execution paths, since it is capable of giving an insight about the influence of an input on the control flow. If it is possible to run to completion an analysis (i.e. it terminates in a feasible time) having made all inputs symbolic, then it is also possible to re-run it again having made all but one inputs symbolic: if there is no difference between the two numbers of obtained paths, then that input, regardless of its value, cannot alter the control flow. Similarly, if only one input has been made symbolic and the number of paths does not change, then that input cannot alter the control flow, given all the concrete values of the other inputs.

Automation of the test harness creation

From the procedure template discussed above it is possible to deduce that generating a test harness for an analysis based on symbolic execution requires a significant manual effort by the tester. This problem has been addressed and has lead to the creation of the KLOVER[19] methodology and framework, capable of automatically generating test harnesses that output unit-level tests with high structural code coverage. KLOVER takes as input a source file and for each function under test generates a test proxy, i.e. a function that wraps the function under test, receives symbolic inputs from the main function of the test harness and performs variables initialization in order to satisfy the preconditions. These preconditions are automatically inferred, as much as possible, with a static analysis on the function under test and they often need to be manually tuned in order to achieve a higher coverage and a lower false positives ratio. This approach simplifies the implementation work (i.e. less code writing) for the tester but it does not diminish the amount of conceptual work needed, since the path explosion and constraints complexity problems are left unaddressed by the framework and must be dealt with manually inside the test proxies.

Chapter 8

Analysis of CAN messages filtering and routing

The analysis of the software modules involved in CAN messages filtering and routing has been initially performed by means of symbolic execution. Since the obtained results are yielded in a form that makes difficult and error-prone to compare them with the specifications, the analysis has been repeated with dynamic taint analysis, in order to further elaborate them automatically. For the second part, both the traditional approach to DTA and the one based on symbolic execution have been used, in order to compare the precision of the results.

8.1 Analysis with symbolic execution

Since the software of the central gateway under analysis is organized into layers with increasing abstraction, the testing approach that initially seemed more appropriate consisted in taking the highest level modules (i.e. the application modules), leaving them untouched and stubbing the interface with the underneath modules.

8.1.1 Content-based Message Filter - Test harness

The first module to be put under analysis has been the Content-based Message Filter: its interface, reported in Listing 8.1, consists in only one function that takes as parameters the identifier of a CAN message and its payload, and returns a boolean value, true if the message shall pass, false otherwise.

```
typedef enum {
    LengthCode_0 = 0,
    LengthCode_1 = 1,
    LengthCode_2 = 2,
    ...
    LengthCode_8 = 8,
    LengthCode_12 = 9,
    LengthCode_16 = 10,
    ...
    LengthCode_64 = 15
} LengthCode_t;
typedef struct {
    void *dataPointer;
    LengthCode_t length;
} MessagePayload_t;
bool ContentBasedMessageFilter_Function(
        unsigned int messageId,
        MessagePayload_t *messagePayload);
```

Listing 8.1. Interface of the Content-based Message Filter.

Stubbing is not required for this module, because it does not use functionalities offered by other modules; its public function is called by the Communication module. The main function of the test harness, reported in Listing 8.2 creates symbolic values for the message identifier, the payload length and the payload buffer. Since the KLEE implementation of the *malloc()* system call cannot take a symbolic value as the size parameter, the payload length must be priorly concretized so that it can take all its allowed concrete values (i.e. it is required to switch from implicit to explicit enumeration); this is feasible in this particular case because the payload length is an enumeration with a very limited set of possible values.

```
int main(void) {
    unsigned int msgId;
    void *dataPtr;
    LengthCode_t length;
    klee_make_symbolic(&msgId, sizeof(msgId), "msgId");
    klee_make_symbolic(&length, sizeof(length), "length");
    if (length == LengthCode_1) length = LengthCode_1;
    else if (length == LengthCode_2) length = LengthCode_2;
    ...
    else if (length == LengthCode_8) length = LengthCode_8;
    ...
    else if (length == LengthCode_64) length = LengthCode_64;
    else length = LengthCode_0;

    if (length > 0) {
        dataPtr = malloc(length);
        klee_make_symbolic(dataPtr, length, "data");
    } else {
        dataPtr = NULL;
    }
    MessagePayload_t payload = {
```

```
                .dataPointer = dataPtr,
                .length = length};
    bool retVal = ContentBasedMessageFilter_Function(msgId,
    payload);
    return (int) retVal;
}
```

Listing 8.2. Main function of the test harness for the Content-based Message Filter.

With the test harness described above, it is possible to explore all the execution paths of the message filter, except for the ones triggered by invalid payload length codes (enumerations in C language are almost always implemented with integer variables, that can take values not listed in the definition) and the ones triggered by mismatches between the declared payload length code and the actual length of the block allocated for payload data (again, due to the properties of the C language, the called has to trust the caller).

8.1.2 Content-based Message Filter - Results

A portion of the result of the analysis is shown in Listing 8.3: for each execution path there is a line containing a sample input message value that leads the execution along that path and a flag that states if the sample input message shall be forwarded; for clarity of explanation, only the paths with message identifier 0x3b4 and length code 6 as path constraint are listed.

```
...
Forward = No;   msgId = 0x000003b4; length = 0x00000006;    data
    = 0x 00 00 00 03 00 00;
Forward = No;   msgId = 0x000003b4; length = 0x00000006;    data
    = 0x 00 00 00 0c 00 00;
Forward = No;   msgId = 0x000003b4; length = 0x00000006;    data
    = 0x 00 00 00 30 00 00;
Forward = No;   msgId = 0x000003b4; length = 0x00000006;    data
    = 0x 00 00 00 c0 00 00;
Forward = Yes;  msgId = 0x000003b4; length = 0x00000006;    data
    = 0x 00 00 00 00 00 00;
...

KLEE: done: total instructions = 1118377
KLEE: done: completed paths = 1424
KLEE: done: generated tests = 1424
```

Listing 8.3. Test results for the Content-based Message Filter.

To interpret the result, it is useful to compare it to the configuration of the same message, reported in Listing 8.4, which includes the identifier, the number of signals inside the message and for each signal, the starting bit, the length, the minimum and the maximum allowed values.

```
{ 0x3B4, 13, {
        {6 , 1, 0x0, 0x1},  {24, 2, 0x0, 0x2},
        {26, 2, 0x0, 0x2},  {28, 2, 0x0, 0x2},
        {30, 2, 0x0, 0x2},  {32, 2, 0x0, 0x3},
        {34, 2, 0x0, 0x3},  {36, 1, 0x0, 0x1},
        {37, 1, 0x0, 0x1},  {38, 1, 0x0, 0x1},
        {39, 1, 0x0, 0x1},  {40, 4, 0x0, 0xF},
        {44, 4, 0x0, 0xF},  }, },
```

Listing 8.4. Configuration of message 0x3b4.

The second, third, fourth and fifth signals are represented on 2 bits, and their value can vary between 0 and 2, so it is possible to have the value 3 which is not allowed and causes the whole message to be blocked. The fifth sample message shall be forwarded because all its signals have in-range values, while all the other sample messages have at least one signal value out-of-range: it is possible to deduct that signal values are checked sequentially and that as soon as an out-of-range value is found, the checking procedure quits blocking the message.

Another portion of the result of the execution is shown in Listing 8.5:

```
...
Forward = Yes;  msgID = 0x000003b4; length = 0x00000000;   data
    = 0x;
Forward = Yes;  msgID = 0x000003b4; length = 0x00000001;   data
    = 0x 00;
...
Forward = Yes;  msgID = 0x000003b4; length = 0x00000030;   data
    = 0x 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00;
...

KLEE: done: total instructions = 1118377
KLEE: done: completed paths = 1424
KLEE: done: generated tests = 1424
```

Listing 8.5. Test results for the Content-based Message Filter.

A message with identifier 0x3b4 must have a payload length of 6 bytes, but messages with different lengths are not blocked, in particular shorter messages that do not contain all the expected signals and that therefore cannot be checked completely. Furthermore, CAN messages have a maximum payload length of 8 bytes, while this message filter analyses and can forward messages with longer payloads, violating the standard. This does not imply that the code is bugged, but that this module relies on other modules for the correctness of its input parameters: specifically, messages longer than 8 bytes shall be discarded at hardware level, while messages whose payload length is different than the expected one given the message identifier shall be discarded by a lower level software module.

The testing approach adopted so far is therefore too naive, because it does not take into account the tight coupling among the software modules and requires the tester to reason about it and to manually constrain the input parameters: without this effort, the false positive ratio explodes and the obtained results are not meaningful. A more effective approach consists in taking all the modules involved in a specific functionality written by the application developers and stubbing the interface with the operating system: this reduces the complexity and the amount of constraints to be applied on input data by the tester, thus it reduces the false positive ratio among the possible bugs.

8.1.3 Messages filtering/routing full stack - Test harness

In the specific case, all the modules involved in message filtering and routing were selected, beginning with the Content-based Message Filter and ending with the CAN Interface, that interfaces with a stub of the operating system's driver for the CAN hardware module. The constraints to be imposed on the input data by the stub are derived only from the CAN standard specifications, not from the architecture of the specific application, which therefore needs not to be studied in detail by the tester. The CAN Interface module provides two public functions, reported in Listing 8.6: the first one is meant to be called during the system boot-up and performs an initialization of the module according to its static configuration, setting in the driver the right number of CAN channels and the allowed message identifiers for each channel; the second one is meant to be called in the context of an Interrupt Service Routine triggered by an incoming message in the CAN hardware module, it performs a check of the payload length (which must correspond to the expected one given the message identifier) and passes the message to the Communication or the CAN Transport Protocol module.

```
void CANInterface_Init(void);
void CANInterface_Receive(unsigned char channelIndex);
```

Listing 8.6. Public functions of the CAN Interface module.

The CAN hardware is organized in modules, nodes and message objects: each module can have multiple nodes, that correspond to physical CAN channels, and each node has an associated list of message objects, that describe the features of the messages allowed to enter and exit the node, the most important of which is the message identifier. In this application, the *CANInterface_Init()* function activates 2 modules, 4 nodes for the first module and 2 nodes for the second, for a total of

6 CAN channels, and creates a total of about two hundred message objects. To do this, it uses the functions in Listing 8.7, which are provided by the operating system's CAN driver.

```c
typedef struct {
    unsigned int moduleId;
} CANDriver_ModuleConfig;
void CANDriver_initModuleConfig(
        CANDriver_ModuleConfig *moduleConfig);
typedef struct {
    unsigned int moduleId;
} CANDriver_Module;
CANDriver_Status CANDriver_initModule(
        CANDriver_Module *module,
        const CANDriver_ModuleConfig *moduleConfig);
typedef struct {
    unsigned int nodeId;
    CANDriver_Module *module;
} CANDriver_NodeConfig;
void CANDriver_initNodeConfig(
        CANDriver_NodeConfig *nodeConfig,
        const CANDriver_Module *module);
typedef struct {
    unsigned int nodeId;
    CANDriver_Module *module;
} CANDriver_Node;
CANDriver_Status CANDriver_initNode(
        CANDriver_Node *node,
        const CANDriver_NodeConfig *nodeConfig);
typedef struct {
    unsigned int msgObjId;
    CANDriver_Node *node;
} CANDriver_MsgObjConfig;
void CANDriver_initMsgObjConfig(
        CANDriver_MsgObjConfig *msgObjConfig,
        const CANDriver_Node *node);
typedef struct {
    unsigned int msgObjId;
    CANDriver_Node *node;
} CANDriver_MsgObj;
CANDriver_Status CANDriver_initMsgObj(
        CANDriver_MsgObj *msgObj,
        const CANDriver_MsgObjConfig *msgObjConfig);
```

Listing 8.7. Functions provided by the CAN driver.

The initialization of modules, nodes, and message objects is implemented by function pairs: first, a configuration object is initialized with the first function, then the CAN Interface sets some parameters (hardware details, identifiers, ...) in the structure, then the actual initialization takes place with the second function, and finally structures representing modules, nodes and message object are returned to the CAN Interface, which has to save them because they are needed for sending and receiving messages. The stub implements all these functions: it discards all the low level hardware parameters and it retains from the configuration objects the information needed to determine the number of the CAN channels and what message identifiers are allowed through each channel. This information allows the stub to discard some invalid packets; this is a necessary feature because this module assumes that the real hardware and driver behave this way.

The functions in Listing 8.8 are provided by the CAN driver and used to send and receive messages.

```
typedef struct {
    unsigned int        messageId;
    unsigned char       data[8];
    LengthCode_t        length;
} CANDriver_Message;
unsigned int CANDriver_getMessageId(void);
CANDriver_Status CANDriver_readMessage(
        CANDriver_Message *msg,
        const CANDriver_MsgObj *msgObj);
CANDriver_Status CANDriver_sendMessage(
        const CANDriver_Message *msg,
        const CANDriver_MsgObj *msgObj);
```

Listing 8.8. Functions provided by the CAN driver.

This module needs to include the message object corresponding to the message it is willing to send or receive, in order to use the correct channel. Actually, the stub implementation of these functions does not need the message object parameter, as all the information needed to use the correct channel can be deduced from the configuration information collected during the initialization phase. Therefore, before reading an incoming message, this module needs to know its identifier, to retrieve the correct message object. This task is accomplished by the *CANDriver-_getMessageId()* function, that checks that the message identifier is among those allowed for the channel, returning the message identifier if true, an error code otherwise, that prevents this module from calling the *CANDriver_readMessage()* function.

The stub provides another function, reported in Listing 8.9, intended to be used in the main function of the test harness.

```
void CANDriver_Stub_TestMessage(
        unsigned int moduleId,
        unsigned int nodeId,
        CANDriver_Message *message);
```

Listing 8.9. Function provided by the CAN driver stub to the main function of the test harness.

This function allows to send a single input CAN message to the desired channel, and to print a log containing the eventual errors detected or a corresponding output message. First, it checks that the payload length is between 0 and 8 bytes, so that it does not violate the CAN specifications; then it determines the correct input channel index given the module and the node ids and if the channel exists, it passes it as parameter to the *CANInterface_Receive()* function, otherwise it logs the error condition and quits.

The main function of the test harness, showed in Listing 8.10, creates symbolic values for every input parameter. Unlike in the test harness for the Content-based Message Filter, here the payload buffer is allocated statically with the maximum length allowed by the CAN specifications.

```
int main(void)
{
    unsigned int moduleId;
    unsigned int nodeId;
    unsigned int messageId;
    unsigned char data[8];
    LengthCode_t length;
    CANDriver_Message message;

    klee_make_symbolic(&moduleId, sizeof(moduleId), "moduleId");
    klee_make_symbolic(&nodeId, sizeof(nodeId), "nodeId");
    klee_make_symbolic(&messageId, sizeof(messageId),"messageId"
   );
    message.messageId = messageId;
    klee_make_symbolic(&length, sizeof(length), "length");
    message.length = length;
    klee_make_symbolic(data, sizeof(data), "data");
    for (unsigned i; i < 8; i++) message.data[i] = data[i];

    CANDriver_Stub_TestMessage(moduleId, nodeId, &message);
    return 0;
}
```

Listing 8.10. Main function of the test harness.

This choice allows to avoid concretizing the payload length in order to be able to perform dynamic memory allocation, and thus to explore the behaviours triggered by invalid length codes (which should be detected by the test harness, as described above); the drawback is that this code is not capable of detecting eventual buffer overflows when the payload length is smaller than 8 bytes. Beside that, all the other possible execution paths through all the involved modules are explored.

8.1.4 Messages filtering/routing full stack - Results

A part of the result of the analysis is reported in Listing 8.11. As with the Content-based Message Filter case, for ease of explanation only the messages with identifier 0x3b4 are reported. The log of each execution path contains the message and the channel that drove the execution towards that path, while the subsequent lines contain an error message if the message was rejected by the stub, or the content of the read message if it was accepted, also with the content of the sent message if it was forwarded.

```
TEST MESSAGE: ID: 0x3b4; Length code: −1; Data: 00 00 00 00 00
   00 00 00; Module: 2; Node: 0;
Bad length code.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 2; Node: 0;
Bad channel.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 1; Node: 6;
Bad channel.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 0; Node: 6;
Bad channel.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 0; Node: 3;
Bad message ID, node ID or module address.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 0; Node: 2;
Bad message ID, node ID or module address.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 0; Node: 1;
Bad message ID, node ID or module address.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 0; Node: 0;
Bad message ID, node ID or module address.

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 1; Node: 1;
```

```
Bad message ID, node ID or module address.

TEST MESSAGE: ID: 0x3b4; Length code: 6; Data: 00 00 00 03 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x000003b4; Length code: 6; Data: 00 00 00 03 00 00 00
    00;

TEST MESSAGE: ID: 0x3b4; Length code: 6; Data: 00 00 00 0c 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x000003b4; Length code: 6; Data: 00 00 00 0c 00 00 00
    00;

TEST MESSAGE: ID: 0x3b4; Length code: 6; Data: 00 00 00 30 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x000003b4; Length code: 6; Data: 00 00 00 30 00 00 00
    00;

TEST MESSAGE: ID: 0x3b4; Length code: 6; Data: 00 00 00 c0 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x000003b4; Length code: 6; Data: 00 00 00 c0 00 00 00
    00;

TEST MESSAGE: ID: 0x3b4; Length code: 0; Data: 00 00 00 00 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x000003b4; Length code: 0; Data: 00 00 00 00 00 00 00
    00;

TEST MESSAGE: ID: 0x3b4; Length code: 6; Data: 00 00 00 00 00 00
    00 00; Module: 1; Node: 0;
READ: ID: 0x3b4; Length code: 6; Data: 00 00 00 00 00 00 00 00;
SEND: ID: 0x3b4; Length code: 6; Data: 00 00 00 00 00 00 00 00;
   Module: 0; Node: 3;
...

KLEE: done: total instructions = 2317405
KLEE: done: completed paths = 1103
KLEE: done: generated tests = 1103
```

Listing 8.11. Results of the analysis of the complete stack of the message filter.

The 1st message was rejected because the length code is an example of out-of-range value that violates the CAN specifications. The 2nd one has an invalid module id, while the 3rd and the 4th an invalid node id, therefore the stub in not able to find an existing input CAN channel. The messages from the 5th to the 9th were inserted in an existing channel, but none of them was allowed to accept a message with that identifier, therefore we get an example for each of the 5 wrong channels. The subsequent 4 messages were inserted in the correct channel, so the CAN Interface module is capable of reading them, but they were not forwarded, because their payloads contain signals with out-of-range values. The second-last message has an incorrect length code, which was not detected by the stub but by the CAN Interface module, which has in its static configuration the associations between message identifier and length code. The last message was correctly read and forwarded to another channel.

All the allowed and invalid messages were checked against the specifications of the functionality of the modules involved in message filtering, and no incongruence nor implementation errors were found by this analysis.

8.2 Analysis with dynamic taint analysis

A physical CAN channel can be seen as a conjunction of a logical input channel and a logical output channel. With this in mind, dynamic taint analysis has been used in this context to to track the information flow between input channels and output channels: the purpose was to observe the influence on the output channels of a certain message inserted in a certain input channel. To do this, the test harness used for the analysis with symbolic execution has been adapted, and both the prototype tool for Multi-path DTA and KLEE-TAINT were employed. The results of this analysis are fundamentally equivalent to those reported in Subsection 8.1.4, its advantage consists in the form in which the results are presented: the obtained test cases are further elaborated in order to deduct the taint propagation, providing a result easier to interpret and to compare with the specifications.

8.2.1 Multi-path DTA - Test harness

The stub of the CAN driver was modified in order to represent input and output channels as arrays indexed by channel number and whose cells contain the message they can carry, and to provide external access to the output channels, as reported in Listing 8.12: sent messages are not logged and printed by the stub, because they need to be examined in the main function of the test harness, before they are concretized in order to be printed. Other minor modifications, like external access to the input channels and the signature modification of the *CANDriver_Stub_TestMessage()*, were done only for simplification purpose and do not affect the analysis.

```
CANDriver_Message *CANDriver_Stub_GetInChannels(void);
CANDriver_Message *CANDriver_Stub_GetOutChannels(void);
bool CANDriver_Stub_TestMessage(uint8 inChannelIdx);
```

Listing 8.12. Functions provided by the CAN driver stub to the main function of the test harness.

The logic behind the main function of the test harness can be subdivided in steps: the first one, shown in Listing 8.13, consists in defining a sample message and the input channel. Having a sample message is not mandatory, but it allows to focus the analysis: for example, it could be interesting to determine the influence on the output channels only of a message with a certain identifier while the content is free to variate, or of a message with a certain content.

```
#define IN_CHANNEL_IDX          4
#define MESSAGE_ID              0x3b4
//#define MESSAGE_LENGTH_CODE       LengthCode_6
//#define MESSAGE_DATA_0            0x00
//#define MESSAGE_DATA_1            0x00
...
//#define MESSAGE_DATA_7            0x00

    CANDriver_Message *inChannels = CANDriver_Stub_GetInChannels
  ();
    CANDriver_Message *outChannels =
  CANDriver_Stub_GetOutChannels();
```

Listing 8.13. Main function of the test harness for analysis with multi-path DTA.

The second step consists in defining the taint source. In Listing 8.14, the whole content of the selected channel is marked as tainted, which includes the message identifier, the payload content and the payload length. The payload content has a fixed size of 8 bytes and all of them are marked as tainted regardless of the length code: the reason is the same that has brought to declare symbolic the whole payload content in the main function of the test harness for symbolic execution.

```
    char taintSourceName[0x100];
    sprintf(taintSourceName, "In channel %u", IN_CHANNEL_IDX);
    s2e_taint_source(&inChannels[IN_CHANNEL_IDX], sizeof(
  inChannels[IN_CHANNEL_IDX]), taintSourceName);
```

Listing 8.14. Main function of the test harness for analysis with multi-path DTA.

The third step consists in defining policies on the taint source, which consists in assigning concrete values or symbolic constrained values to parts of the taint source. This step is optional. In Listing 8.15, since only the *MESSAGE_ID* macro is defined, only the message identifier is forced to have a single value: the analysis reports what output channels are influenced by a message with identifier 0x3b4 injected into input channel 4.

```
#ifdef MESSAGE_ID
    inChannels[IN_CHANNEL_IDX].id = MESSAGE_ID;
#endif
#ifdef MESSAGE_LENGTH_CODE
    inChannels[IN_CHANNEL_IDX].lengthCode = MESSAGE_LENGTH_CODE;
```

```
#endif
#ifdef MESSAGE_DATA_0
    inChannels[IN_CHANNEL_IDX].data[0] = MESSAGE_DATA_0;
#endif
#ifdef MESSAGE_DATA_1
    inChannels[IN_CHANNEL_IDX].data[1] = MESSAGE_DATA_1;
#endif
...
#ifdef MESSAGE_DATA_7
    inChannels[IN_CHANNEL_IDX].data[7] = MESSAGE_DATA_7;
#endif
```

Listing 8.15. Main function of the test harness for analysis with multi-path DTA.

The fourth step consists in defining the taint sinks, after having run the procedure under test. In Listing 8.16, all the output channels are defined as taint sinks, since it has been decided to observe the influence of the input message on all of them.

```
    CANDriver_Stub_TestMessage(IN_CHANNEL_IDX);

    char taintSinkName[0x100];
    for (uint16 i = 0; i < CANDRIVER_STUB_CHANNELS_MAX_NUM; i++)
    {
        sprintf(taintSinkName, "Out channel %u", i);
        s2e_taint_sink(&outChannels[i], sizeof(outChannels[i]),
  taintSinkName);
    }
```

Listing 8.16. Main function of the test harness for analysis with multi-path DTA.

The last step consists in observing the content of all the channels and it is optional. The purpose is to clarify why a certain output channel is tainted and what messages cause this, which is equivalent to getting a test case with the test harness described in Subsection 8.1.3. The code is omitted as it is trivial.

8.2.2 Multi-path DTA - Results

Part of the result of the analysis performed with the described test harness is reported in Listing 8.17. Before the end of the execution of each state, the taint status of all the declared taint sinks (i.e. the output channels) is observed. The content of the channels is not reported for simplicity, as it is the same of the test cases reported in Subsection 8.1.4.

```
[State 0] In channel 4'; Addr 0x80610c0; Size 0x10;

[State 0] Out chn 0: Addr 0x8061100; Size 0x10; Status
   NOT_TAINTED;
```

```
[State 0] Out chn 1: Addr 0x8061110; Size 0x10; Status
    NOT_TAINTED;
[State 0] Out chn 2: Addr 0x8061120; Size 0x10; Status
    NOT_TAINTED;
[State 0] Out chn 3: Addr 0x8061130; Size 0x10; Status
    NOT_TAINTED;
[State 0] Out chn 4: Addr 0x8061140; Size 0x10; Status
    NOT_TAINTED;
[State 0] Out chn 5: Addr 0x8061150; Size 0x10; Status
    NOT_TAINTED;
...
[State 1] Out chn 0: Addr 0x8061100; Size 0x10; Status
    NOT_TAINTED;
[State 1] Out chn 1: Addr 0x8061110; Size 0x10; Status
    NOT_TAINTED;
[State 1] Out chn 2: Addr 0x8061120; Size 0x10; Status
    NOT_TAINTED;
[State 1] Out chn 3: Addr 0x8061130; Size 0x10; Status TAINTED;
[State 1] Out chn 4: Addr 0x8061140; Size 0x10; Status
    NOT_TAINTED;
[State 1] Out chn 5: Addr 0x8061150; Size 0x10; Status
    NOT_TAINTED;
...
[State 6] Out chn 0: Addr 0x8061100; Size 0x10; Status
    NOT_TAINTED;
[State 6] Out chn 1: Addr 0x8061110; Size 0x10; Status
    NOT_TAINTED;
[State 6] Out chn 2: Addr 0x8061120; Size 0x10; Status
    NOT_TAINTED;
[State 6] Out chn 3: Addr 0x8061130; Size 0x10; Status TAINTED;
[State 6] Out chn 4: Addr 0x8061140; Size 0x10; Status
    NOT_TAINTED;
[State 6] Out chn 5: Addr 0x8061150; Size 0x10; Status
    NOT_TAINTED;
```

Listing 8.17. Results of the analysis of the complete stack of the message filter with multi-path DTA.

The Multi-path DTA tool observes that the contents of the taint sinks in states 0 and 1 are not equal, thus it changes the tainting status of output channel 3 to tainted. At the end of the analysis only channel 3 results tainted: the tool has exhaustively checked that a message with identifier 0x3b4 inserted into channel 4 can only influence channel 3, regardless of the size and the content of its payload. This analysis has been repeated for all channels and unconstraining the message identifier: again, the obtained results were coherent with the specifications for the CAN message filtering and routing functionality.

The execution engine generates a total of 7 execution states, while the test harness in Subsection 8.1.3 generated 15 states: this happens due to the fact that the former test harness constrains the channel index to a single concrete value and due to the difference between the interfaces of the stub for the CAN driver (respectively Listing 8.12 and Listing 8.9). In the former, input channels are grouped into an array of 6 elements and identified by means of an index (0 to 5). In the latter, they are selected by means of a combination of a CAN module identifier (0 to 1) and a CAN node identifier (0 to 5). Each module has its own nodes, but all the nodes are uniquely identified. Specifying a node identifier is not really necessary, but it allows a more exhaustive analysis, since it tests the capability of the CAN Interface module to detect and discard invalid combinations. There are 9 possible combinations of module and node identifiers, as shown in Figure 8.1: 6 of them correspond to valid input channels, 2 are invalid because the module exist but the node does not, 1 is invalid because the module does not exist (then the node is not checked). The test harness in Subsection 8.1.3 explores all these combinations, forking 9 paths, while the one in Subsection 8.2.1 only the one corresponding to the provided concrete index, hence the difference of 8 execution paths.

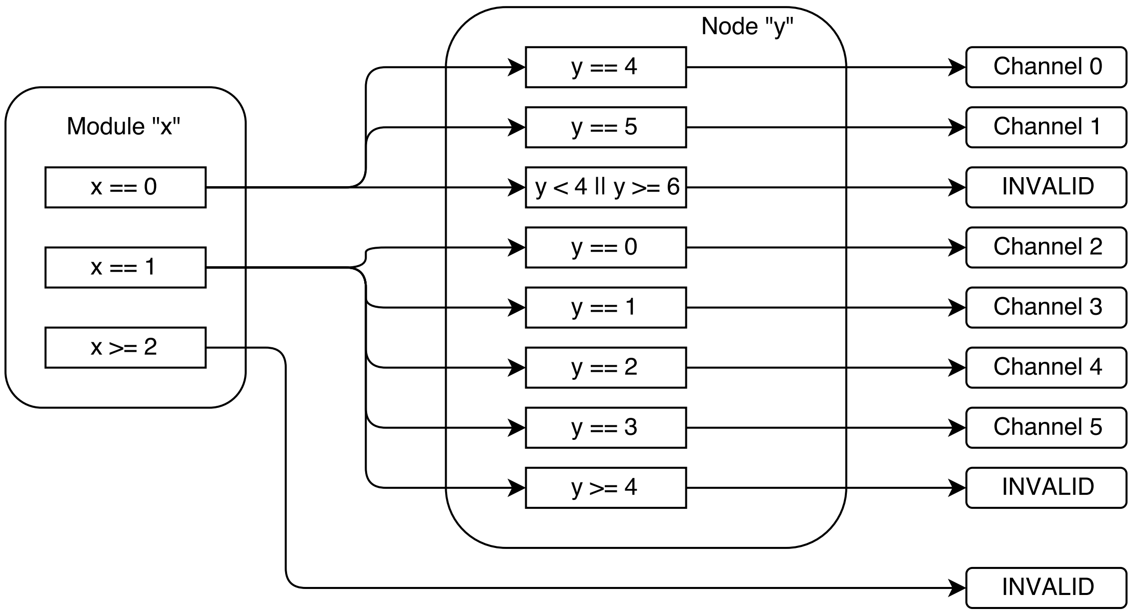

Figure 8.1. Possible combinations of module and node identifiers.

8.2.3 Traditional DTA

The analysis described in Subsection 8.2.1 and in Subsection 8.2.2 has been repeated with KLEE-TAINT, that implements the traditional approach to dynamic taint analysis, in order to verify the improvement in effectiveness introduced by the Multi-path approach. The test harness had to be adapted for interfacing with the functions of KLEE-TAINT that allow to define taint sources and taint sinks. The order of taint sources definition and taint source policies enforcement phases had to be inverted with respect to the procedure described before: the reason of this is that enforcing a policy consists in assigning a constant value to (a part of) a taint source, which resets its taint value, therefore it is necessary first to assign a content value, then to taint it. The tool has been configured to run first in direct mode then in region-based mode in order to maximize the precision, but the results were not affected. The printing format is similar to the one used in Subsection 8.2.2: the input channel chosen as taint source is printed together with its taint value, which has no analogous in multi-path DTA, all the output channels are printed with their taint value in place of the taint status, which is assumed to be positive if the taint value is different from 0. In addition, since no symbolic values are used, the concrete values given to the input channels and those taken by the output channels are reported, for ease of explanation.

Since the result of the analysis depends on the value of the taint source, for the first step the input channel 4 was assigned a message that the gateway forwards to the output channel 3, i.e. which has a message identifier allowed for that channel, and payload size and content coherent with the identifier. As reported in Listing 8.18, KLEE-TAINT is able to propagate correctly the taint associated to input channel 4 to output channel 3.

```
In chn 4: Addr 0x2bf56d0; Size 0x00000010; Taint 0x00000001;
Out chn 0: Addr 0x2bf5690; Size 0x00000010; Taint 0000000000;
Out chn 1: Addr 0x2bf56a0; Size 0x00000010; Taint 0000000000;
Out chn 2: Addr 0x2bf56b0; Size 0x00000010; Taint 0000000000;
Out chn 3: Addr 0x2bf56c0; Size 0x00000010; Taint 0x00000001;
Out chn 4: Addr 0x2bf56d0; Size 0x00000010; Taint 0000000000;
Out chn 5: Addr 0x2bf56e0; Size 0x00000010; Taint 0000000000;
Out chn 0: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
Out chn 1: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
Out chn 2: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
Out chn 3: id = 0x000003b4; lengthCode = 0006; data = 01 01 01
    01 01 01 00 00;
Out chn 4: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
Out chn 5: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
In chn 0: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
```

```
In chn 1: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 2: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 01 01 01 01
    01 01 00 00;
In chn 5: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
```

Listing 8.18. Results of the analysis of the complete stack of the message filter with KLEE-TAINT.

The second step consisted in assigning the input channel 4 a message almost identical to the one of the first step, the only difference being an out-of-range value of one of the signals in the payload, which prevents the message from being forwarded. In Listing 8.19, output channel 3 should be tainted with value 1, but KLEE-TAINT does not determine its tainting status correctly: the tainting status of a taint sink should not depend on a particular content of the taint source, if a variation of the content of a taint source causes a variation of the content of a taint sink, then the latter is tainted.

```
In chn 4: Addr 0x2bf56d0; Size 0x00000010; Taint 0x00000001;
Out chn 0: Addr 0x2bfd410; Size 0x00000010; Taint 0000000000;
Out chn 1: Addr 0x2bfd420; Size 0x00000010; Taint 0000000000;
Out chn 2: Addr 0x2bfd430; Size 0x00000010; Taint 0000000000;
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0000000000;
Out chn 4: Addr 0x2bfd450; Size 0x00000010; Taint 0000000000;
Out chn 5: Addr 0x2bfd460; Size 0x00000010; Taint 0000000000;
Out chn 0: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
Out chn 1: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
Out chn 2: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
Out chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
Out chn 4: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
Out chn 5: id = 0000000000; lengthCode = 0000; data = 00 00 00
   00 00 00 00 00;
In chn 0: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 1: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
```

```
In chn 2: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 01 01 01 c1
    01 01 01 01;
In chn 5: id = 0000000000; lengthCode = 0000; data = 00 00 00 00
    00 00 00 00;
```

Listing 8.19. Results of the analysis of the complete stack of the message filter with KLEE-TAINT.

8.2.4 Enhanced traditional DTA

Since KLEE-TAINT is capable of performing symbolic execution, it is possible to use this feature to increment the precision of taint propagation. Considering the same example of Subsection 8.2.3, it is sufficient to declare the payload content symbolic instead of assigning it a constant value. The printing format is the same used in Subsection 8.2.3, the only differences being the omission of information related to non-involved channels and the fact that the content values of the channels are not manually assigned but automatically generated by concretizing symbolic expressions.

As shown in Listing 8.20, the tool now creates 5 execution states, triggered by the logic of the Content-based Message Filter: the corresponding test cases are equivalent to the ones reported in Subsection 8.1.2. These results are equivalent to the ones in Subsection 8.2.2: in the last state the taint was propagated correctly, which is sufficient to determine that output channel 3 is tainted.

```
In chn 4: Addr 0x2bfd450; Size 0x00000010; Taint 0x00000001;

...
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0000000000;
...
Out chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
...
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 00 00 00 03
    00 00 00 00;
...

...
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0000000000;
...
Out chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
```

```
...
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 00 00 00 0c
    00 00 00 00;
...

...
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0000000000;
...
Out chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
...
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 00 00 00 30
    00 00 00 00;
...

...
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0000000000;
...
Out chn 3: id = 0000000000; lengthCode = 0000; data = 00 00 00
    00 00 00 00 00;
...
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 00 00 00 c0
    00 00 00 00;
...

...
Out chn 3: Addr 0x2bfd440; Size 0x00000010; Taint 0x00000001;
...
Out chn 3: id = 0x000003b4; lengthCode = 0006; data = 00 00 00
    00 00 00 00 00;
...
In chn 4: id = 0x000003b4; lengthCode = 0006; data = 00 00 00 00
    00 00 00 00;
...
```

Listing 8.20. Results of the analysis of the complete stack of the message filter with KLEE-TAINT, using symbolic execution.

8.2.5 Enhanced traditional DTA - Limitations

It is important to underline that the results in Subsection 8.2.2 and in Subsection 8.2.4 are equivalent only in this particular case: in general, the traditional taint propagation approach enhanced with symbolic execution does not achieve the same taint precision level of multi-path DTA. It is possible to show this by analysing the same example discussed in Section 3.4, reported in Listing 8.21, where the instrumentation code is omitted for simplicity.

```
int a;
int b;

/* Define "a" as taint source */

if (a == 5) {
    b = sqrt(256);
} else {
    b = pow(4, 2);
}

/* Define "b" as taint sink */
```

Listing 8.21. Example that causes traditional taint propagation enhanced with symbolic execution to fail.

The multi-path DTA prototype correctly deduces that variable b is not tainted, as shown in Listing 8.22, while KLEE-TAINT (with taint source a declared symbolic) assigns the same taint value of the source to the sink in both execution paths, as shown in Listing 8.23. The printing format of the results obtained with KLEE-TAINT is here simplified: *In taint* is the taint value assigned to the source a, while *Out taint* is the equivalent for the sink b; after the first line, one line is printed for each execution state, containing the taint value of the sink and sample concrete values for the source and the sink.

```
[State 0] Name = 'a'; Address = 0xbfacb200; Size = 0x4;

[State 0] Name = 'b'; Address = 0xbfacb204; Size = 0x4; Status =
    NOT_TAINTED;
[State 0] b = 16; a = 0;

[State 1] Name = 'b'; Address = 0xbfacb204; Size = 0x4; Status =
    NOT_TAINTED;
[State 1] b = 16; a = 5;
```

Listing 8.22. Results of the analysis with multi-path DTA of the example in Listing 8.21.

```
In taint = 1

Out taint = 1; b = 16; a = 0;

Out taint = 1; b = 16; a = 5;
```

Listing 8.23. Results of the analysis with KLEE-TAINT of the example in Listing 8.21.

This result might lead to think that if the taint value of the sink is the same across all execution paths, then the sink is to be considered not tainted, but it is sufficient to slightly modify the example in Listing 8.21 to prove this wrong, as shown in Listing 8.24.

```
int a;
int b;

/* Define 'a' as taint source */

if (a == 5) {
    b = sqrt(256);
} else if (a == 4) {
    b = pow(4, 2);
} else {
    b = 3;
}

/* Define 'b' as taint sink */
```

Listing 8.24. Example that causes traditional taint propagation enhanced with symbolic execution to fail.

Now the execution forks into 3 paths, in 2 of which the taint sink b contains the value 16, and in the remaining one the value 3. The Multi-path DTA tool correctly determines that the sink this time is tainted, as reported in Listing 8.25, while KLEE-TAINT behaves as before, assigning the taint value of the source to the sink in all execution paths, as reported in Listing 8.26.

```
[State 0] Name = 'a'; Address = 0xbfa430c8; Size = 0x4;

[State 0] Name = 'b'; Address = 0xbfa430cc; Size = 0x4; Status =
    NOT_TAINTED;
[State 0] b = 3; a = 0;

[State 2] Name = 'b'; Address = 0xbfa430cc; Size = 0x4; Status =
    TAINTED;
[State 2] b = 16; a = 4;

[State 1] Name = 'b'; Address = 0xbfa430cc; Size = 0x4; Status =
    TAINTED;
[State 1] b = 16; a = 5;
```

Listing 8.25. Results of the analysis with multi-path DTA of the example in Listing 8.24.

```
In taint = 1

```

```
Out taint = 1; b = 3; a = 0;

Out taint = 1; b = 16; a = 4;

Out taint = 1; b = 16; a = 5;
```

Listing 8.26. Results of the analysis with KLEE-TAINT of the example in Listing 8.24.

To conclude, nor the absolute taint values of a sink nor the differences among them are sufficient to deduce its taint status, the differences among the content values are.

Chapter 9

External diagnostic devices authentication

Differently from the analysis described in Chapter 8, the main purpose of this one is not to verify the compliance with detailed specifications, but to discover memory errors. Since the authentication procedure requires the parsing of a public key certificate and the verification of its emitter, it naturally lent itself to an analysis based on symbolic execution, because the control flow is heavily influenced by externally injected data under the control of a potential attacker. While it is theoretically possible to employ also dynamic taint analysis, it is practically infeasible: it would be necessary to statically identify all the memory locations involved in the procedure and declare everything else a taint sink, so that when an illegal access to the memory is performed, the taint status of the non-involved locations becomes positive. Relying on the capabilities of LLVM and KLEE of identifying illegal accesses requires a significantly inferior effort in designing and building the test harness.

The list of the vulnerabilities found with this analysis are listed and briefly described in Table 9.1. Their detailed description, which includes, for each one of them, the affected function and line, a bug-triggering certificate and the explanation of the mechanism, is reported from Section 9.3 to Section 9.6.

9.1 Stub of the driver for the cryptographic accelerator

As discussed in Section 6.2, external diagnostic devices authentication relies on hardware-accelerated asymmetric cryptography. Since the hardware of the ECU under test was not available, a possible solution could have been to use a publicly available software implementation of the required functionalities, wrapped so that its interface would have been the same of the operating system's driver for the

#	Affected module	Description
1	X.509 Certificate Parser - 1st ver.	Integer elements are always assumed to have at least one value byte.
2	X.509 Certificate Parser - 1st ver.	Multi-part tags are not parsed correctly.
3	X.509 Certificate Parser - 2nd ver.	The capability of parsing multi-part tags is absent.
4	X.509 Certificate Parser - 2nd ver.	Certificate serial's content is accessed before checking its length.
5	X.509 Certificate Parser - 2nd ver.	Certificate version's content is accessed before checking its length. The version may take invalid values.
6	X.509 Certificate Parser - 2nd ver.	The bytes after the first of long-form lengths are not parsed correctly.
7	X.509 Certificate Parser - 2nd ver.	The first byte of long-form lengths is not parsed correctly.
8	X.509 Certificate Parser - 2nd ver.	Access to non-existent tag encoding byte.
9	X.509 Certificate Parser - 2nd ver.	Access to non-existent certificate version bit-string.
10	X.509 Certificate Parser - 2nd ver.	Element's hierarchy exceeds the maximum depth.
11	X.509 Certificate Parser - 2nd ver.	Certificate signature's content is accessed before checking its length.
12	X.509 Certificate Parser - 2nd ver.	Signing algorithm and signed algorithm sections have different lengths, but they are compared regardless.
13	X.509 Certificate Parser - 2nd ver.	Out-of-bounds month value is used to index an array.
14	X.509 Certificate Parser - 2nd ver.	Out-of-bounds month value is used to index an array.
15	Certificate Manager - X.509 certificate validation	Certificate issuer' length is not checked. This allows a theoretical side-channel attack.

Table 9.1. List of the vulnerabilities found in the procedure for external diagnostic devices authentication.

cryptographic accelerator. However symbolic execution, and techniques based on symbolic execution such as multi-path DTA, are unable to explore software modules that generate too hard to solve constraints, specifically they are completely unsuitable for exploring cryptographic routines. In the example reported in Listing 9.1, adapted from [20], in order to take the true branch of the if-then-else statement, the symbolic execution engine has to verify its feasibility. To do so, it queries the constraints solver, which has to invert a cryptographic hash function. Since these functions are purposely designed to make it impossible in a feasible time, the whole analysis gets stuck. Similar considerations apply to routines that perform symmetrical and asymmetrical cryptography.

```c
char buffer[SIZE];
... // Fill "buffer".
if (0x12345678 == sha1(buffer, SIZE) {
    printf("OK\n");
} else {
    printf("NO\n");
}
```

Listing 9.1. Example of code using a cryptographic routine that generates too hard to solve constraints. .

Therefore, the first step for building a test harness for the modules involved in authentication has been to stub the aforementioned driver: the stub makes no use of the input parameters, while each output parameter and the return value are assigned completely unconstrained symbolic values. An alternative solution could have been to substitute the driver with symbolic functions[20], that abstract the original implementation with properties such as that decryption inverts decryption. However, the described stub is trivial to implement and, for this analysis, the under-constraining that introduces does not affect the results.

The *CryptoDriver_VerifyToBeSigned()* function (Listing 9.2) validates the to-be-signed section of a X.509 certificate: it takes as input parameters the exponent and the modulo of the public key, the section, the signature and two flags, while the only output parameter is the pointer to a result flag stating the validity of the certificate. The implementation simply assigns an unconstrained symbolic value to the result flag.

```c
void CryptoDriver_VerifyToBeSigned(
        const unsigned char* Exp,    unsigned int ExpSize,
        const unsigned char* Mod,    unsigned int ModSize,
        unsigned char KeyIsString,
        const unsigned char* Msg,    unsigned int MsgSize,
        const unsigned char* Sig,    unsigned int SigSize,
        unsigned char* Result,
        unsigned char MsgIsString)
{
    klee_make_symbolic(Result, sizeof(*Result), "
  VerifyToBeSigned_Result");
    return;
}
```

Listing 9.2. Function of the driver of the cryptographic accelerator used to verify the to-be-signed section of a X.509 certificate.

CryptoDriver_DecryptRSABlock() (Listing 9.3) is the other function of the driver needed by the authentication modules. It decrypts an RSA block and is used to verify a challenge: it takes as input parameters the exponent and the modulo of the public key, the encrypted block and two flags, while the output parameter is the decrypted block and the return value signals errors like wrong sizes. The implementation returns an unconstrained return value and assigns the decrypted block 256 bytes of unconstrained symbolic data, which is the maximum block size of RSA with a 2048 bit key. Such a large symbolic array does not cause problems to the constraint solver for this particular analysis, since the decrypted block is only compared for equality with the original random number, therefore the generated constraints consist in one independent equality condition for each byte.

```
CryptoDriver_ReturnValue_t CryptoDriver_DecryptRSABlock(
        const unsigned char* Exp,   unsigned int ExpSize,
        const unsigned char* Mod,   unsigned int ModSize,
        uint8_t          KeyIsString,
        const unsigned char* CipherIn,   unsigned int
  CipherInSize,
        unsigned char* MsgOut,   unsigned int* MsgOutSize,
        uint8_t          MsgIsString)
{
    CryptoDriver_ReturnValue_t returnValue;
    klee_make_symbolic(MsgOut, sizeof(*MsgOut) * 256, "
  DecryptRSABlock_MsgOut");
    klee_make_symbolic(&returnValue, sizeof(returnValue), "
  DecryptRSABlock_returnValue");
    return returnValue;
}
```

Listing 9.3. Function of the driver of the cryptographic accelerator used to decrypt a RSA block.

Describing these choices with the terminology of S2E, the modules involved in authentication take the role of the unit while the driver for the cryptographic hardware accelerator takes the role of the environment, and the test harness implements a form of Local or Over-approximate execution consistency model: the output parameters and the return value of the functions above have no constraints by interface contract. The main difference with respect to the definition of these models is that the module under test is left untouched and the creation of symbolic data happens inside the environment, but this has no real effect on the analysis.

9.2 Modules selection and main function of the test harness

The testing approach initially selected for the modules involved in authentication is similar to the one used for the modules involved in message filtering, i.e. to stub the driver for the CAN hardware module and inject a sequence of symbolic CAN frames whose identifier is constrained to belong to a subset of identifiers that cause the message to be redirected to the CAN Transport Protocol module by the CAN Interface module. This requires no substantial modification of the test harness for analysis with symbolic execution of the complete stack of the message filter. However this rapidly proved to be completely infeasible. A CAN frame payload directed to the CAN Transport protocol is interpreted as a ISO-TP segment, and if the frame payload is completely symbolic then also the frame type, the message length, and eventually the segment index are symbolic, and this is a source of path explosion: for instance, after a consecutive frame containing part of a large message could come another frame of any type, which could contain the subsequent segment of the same message, a non-contiguous segment, an already received segment, a single frame, the first frame of a new message or a subsequent frame of another large message with a size that can take any value between 0 and 4096 bytes, and in all these cases but the first, the current message has to be discarded. The analysis remains thus stuck in the CAN Transport Protocol module, it is necessary an infeasibly large amount of time to recompose a message able to reach a path through all the other modules. This required a modification of the stub of the driver for the CAN hardware module: *CANDriver_Stub_TestTransportProtocol()* (Listing 9.4) is a function that takes as input parameter a (large) message packet and sends a sequence of CAN messages to the CAN Interface module was added to the interface, intended to be used in the main function of the test harness. The function internally subdivides the message packet in a sequence of segments according to the ISO-TP specifications, the segments being of single frame or multi frame depending on the length of the message packet. Making symbolic only the content of the message packet and/or its length makes symbolic only the payload of the segments and/or its length in the first frame, allowing to reach quickly execution paths of other modules and thus effectively taming the path explosion problem, at the cost of not exploring exhaustively all the possible behaviours of the CAN Transport Protocol module.

```
void CANDriver_Stub_TestTransportProtocol(
        unsigned int moduleId,
        unsigned int nodeId,
        unsigned int messageId,
        unsigned char *messagePacket,
        size_t messagePacketLength);
```

Listing 9.4. Function of the driver of the CAN hardware module used to bypass the CAN Transport Protocol module.

However this simplification proved to be insufficient to explore even a single execution path through the Diagnostic Protocol module. Manual exploration of the code of the CAN Transport Protocol, in order to further constrain the input data and thus simplify the analysis, was infeasible due to the fact that its code is mostly auto-generated and not documented. The test harness could then directly interface with the Diagnostic Protocol module, whose interface consists in 2 callbacks, reported in Listing 9.5. As discussed in Section 6.2, these functions implement the two-phases authentication procedure, but only at a high level: they delegate the verification of the certificate and of the challenge, and the generation of a random number to another module running on the HSM, thus they do no not directly elaborate any of their input and output parameters. In other words, they do not add other relevant execution paths to those already generated by the modules they rely on, in an analysis based on symbolic execution they are "transparent". The same observation applies to the Host to HSM and HSM to Host modules, as their only purpose is to pass requests and data buffers without elaborating them.

```
typedef DPM_GenericData_t unsigned char;
unsigned char DPM_Callback_VerifyX509_and_GetRnd(
        const DPM_GenericData_t *certificateBuffer,
        DPM_GenericData_t *plainRandNum,
        DPM_ErrorCode_t *errorCode);
unsigned char DPM_Callback_VerifyChallenge(
        const DPM_GenericData_t *encryptedRandNum);
```

Listing 9.5. Interface of the Diagnostic Protocol module.

The approach that has been finally adopted consists in two steps: first, to focus the analysis on both versions of the X.509 Certificate Parser, which performs a "fine-grained" elaboration on its input parameters and thus it is suitable for analysis based on symbolic execution; second, to extend the analysis to the whole Certificate Manager and to the modules it relies on, i.e. the Certificate Parser (whose input parameters become more constrained) and the Cryptographic Manager, which relies on the stub described in Section 9.1. The interfaces of these two modules are reported in Listing 9.6. The *X509CertParser_Parse()* function receives a X.509 certificate in ASN.1 DER binary format and fills a structure received as first parameter: the largest fields like public key and signature, and the fields with variable length like issuer and subject are not copied into the structure, their offset and size in the binary are recorded instead; this is due to the fact that the parsed certificate structure is meant to be allocated in the stack by the Certificate Manager module. The *CertificateManager_SetCertificateContext()* and *CertificateManager-_SetChallengeContext()* functions allow to copy the data necessary for each of the two authentication phases, while *CertificateManager_Authenticate()* executes the specified phase and it is meant to be called immediately after each one of the previous two.

```
X509CertParser_ErrorCode_t X509CertParser_Parse(
        X509CertParser_X509Certificate_t     *parsedCertificate,
        const unsigned char                  *certificateBuffer,
        const size_t                         certificateSize);

void CertificateManager_SetCertificateContext(
        const unsigned char      *certificateBuffer,
        const size_t             certificateSize,
        unsigned char            *resultFlag);
void CertificateManager_SetChallengeContext(
        const unsigned char      *encryptedRandNumBuffer,
        const size_t             encryptedRandNumSize,
        const unsigned char      *plainRandNumBuffer,
        const size_t             plainRandNumSize,
        unsigned char            *resultFlag);
void CertificateManager_Authenticate(
        const unsigned char      phaseNumber);
```

Listing 9.6. Interfaces of the X.509 Certificate Parser and Certificate Manager modules.

For the first step, the main function of the test harness instantiates a certificate structure and passes to the parsing function an unconstrained symbolic buffer with a fixed size, as shown in Listing 9.7. The certificate length has been initially fixed to 9 bytes, in order to focus the analysis on the sub-functions that parse the first fields of a X.509 certificate, which are version and algorithm.

```
#define CERT_LEN
unsigned char certBuf[CERT_LEN];

klee_make_symbolic(certBuf, CERT_LEN, "certBuf");

X509CertParser_X509Certificate_t cert;
X509CertParser_ErrorCode_t err =
        X509CertParser_Parse(&cert, certBuf, CERT_LEN);
...
```

Listing 9.7. Main function of the test harness for testing the X.509 Certificate Parser module.

9.3 X.509 Certificate Parser - First version

1. The *X509CertParser_ParseInt()* function (Listing 9.8) parses elements of integer type, copying their content to a C integer variable. In line 9, it checks if the integer number is positive or negative by reading the most significant bit of the first byte of the value, without verifying before that its length is at least one byte: integer elements are assumed to always have a value. Therefore, if the element to be parsed has length 0 and it is the last element in the certificate buffer, the function attempts to read a memory location that is not part of the binary of the certificate: in the best case it simply reads a casual value and the whole parsing function terminates with an error condition for other reasons, in the worst case it may cause a segmentation fault, depending on the CPU architecture and on the operating system.

```
 1 X509CertParser_ErrorCode_t X509CertParser_ParseInt(
 2         const X509CertParser_Token_t *element,
 3         int *value) {
 4     unsigned char isNegative;
 5     const unsigned char *data = element->data;
 6     if (element->length > sizeof(*value)) {
 7         return ERRORCODE_ERROR_MEMORY;
 8     }
 9     if (*data & 0x80) {
10         negative = TRUE;
11         *value = *data & 0x7F;
12     } else {
13         negative = FALSE;
14         *value = *data;
15     }
16     for (data += 1; data < element->data + element->length;
   data++) {
17         *value = (*value << 8) | *data;
18     }
19     if (negative == TRUE) {
20         *value = *value * -1;
21     }
22     return ERRORCODE_OK;
23 }
```

Listing 9.8. Function for parsing integer values: it accesses an element's first octet before checking its length.

When the certificate shown in Listing 9.9 is given as input, *X509CertParser-_ParseInt()* causes an out-of-bounds memory error. The first element is a sequence that includes all the fields of the certificate. The second element is a sequence containing the to-be-signed section of the certificate. The third element is the certificate version section, while the fourth element, containing the actual value of the certificate version, is optional: if present, it specifies if the X.509 certificate is version 2 or 3; otherwise, the certificate is assumed to be version 1. Therefore, if the fourth element is present, it must have a value, its absence should be detected and considered as an error condition.

```
unsigned char certificateBinary[0x8] = {
    0x30,   // Type: universal, constructed, sequence;
    0x06,   // Length: definite short, 6 octets value;
        0x30,   // T: universal, constructed, sequence;
        0x04,   // L: definite short, 4 octets value;
            0xa0,   // T: context specific, constructed;
            0x02,   // L: definite short, 2 octets value;
                0x02,   // T: universal, primitive, integer;
                0x00    // L: definite short, 0 octets value;
};
```

Listing 9.9. Certificate triggering Vulnerability 1: the fourth element is assumed to always have a value.

2. The *X509CertParser_Next()* function (Listing 9.10) parses the tag and the length of each element, filling the corresponding fields inside the *token* structure. Lines 12, 13 and 14 isolate the fields of the first octet of the tag encoding, pointed by *parser->current*. Then, if the tag number is 31, the actual tag number is recomposed from the subsequent octets: the current tag is shifted left by 7 bits, the 7 least significant bits of the current octet are attached to it, the current pointer to the certificate buffer is incremented by 1 and finally the most significant bit of the current octet is checked. The current pointer is incremented (line 22) before checking the most significant bit (line 27), while these two operations should be performed in reverse order.

```
1  X509CertParser_ErrorCode_t X509CertParser_Next(
2          X509CertParser_Parser_t *parser) {
3  #define INCREMENT_CURRENT do { \
4          parser->current++; \
5          if (parser->current >= parser->parents[parser->depth]) { \
6              return ERRORCODE_ERROR_INVALID; \
7          } \
8      } while (0)
9      X509CertParser_Token_t *token = parser->token;
10     ...
11     Util_MemSet(token, 0, sizeof(*token));
12     token->class = (((*parser->current) & (3 << 6)) >> 6);
13     token->isPrimitive = (((*parser->current) & (1 << 5)) == 0);
```

```
    token->tag = *parser->current & ((1 << 5)-1);
    INCREMENT_CURRENT;
    if (token->tag == 31) {
        unsigned int bits = 0;
        token->tag = 0;
        do {
            token->tag <<= 7;
            token->tag |= *parser->current & ((1 << 7) - 1);
            INCREMENT_CURRENT;
            bits += 7;
            if (bits > sizeof(token->tag) * 8) {
                return ERRORCODE_ERROR_MEMORY;
            }
        } while (*parser->current & 0x80);
    }
    ...
```

Listing 9.10. Function for parsing tag and length of an element: the current pointer is incremented before checking the "more" bit.

The certificate shown in Listing 9.11 is similar to the one in Listing 9.9, the only difference consisting in the tag encoding of the third element, which is now multi-part, because the tag number of the first octet is 31. *X509CertParser-_Next()* considers the two following octets as part of this tag, despite the fact that the first bit of the second octet is set to zero, thus the third octet is actually part of the length encoding. More in detail, during the first iteration of the do-while cycle, after the part of the tag number contained in the second octet has been successfully read, the current pointer is incremented and the most significant bit of the third octet is checked in place of the second. During the second and last iteration, the most significant bit of the first octet of the length encoding is checked instead of the third octet of the tag encoding. The parser accepts this certificate as semantically equivalent to the one in Listing 9.9, it triggers the same buffer overflow memory error in the same line of code, while the error condition should be detected and cause a return with error before the function that parses integer elements is called. This bug may prevent valid certificates from being correctly parsed and validated and might allow incorrectly encoded certificates to pass the parsing phase, but it does not compromise in any way the signature verification.

```
unsigned char certificateBinary[0xA] = {
    0x30,    // T: universal, constructed, sequence;
    0x08,    // L: definite short, 8 octets value;
        0x30,    // T: universal, constructed, sequence;
        0x06,    // L: definite short, 6 octets value;
            0xbf,0x00,0x80, // T: context-specific,
   constructed, non well-formed multi-part tag;
```

```
                0x02,           // L: definite short, 2 octets
    value;
                     0x02,    // T: universal, primitive, integer;
                     0x00     // L: definite short, 0 octets value
    ;
};
```

Listing 9.11. Certificate triggering Vulnerability 2: the multi-part tag of the third element is not well formed.

9.4 X.509 Certificate Parser - Second version

3. The second version of the certificate parser "addresses" the problem described in Vulnerability 2 by removing the capability of recognizing multi-part tags, as shown in Listing 9.12: the octets following the first of the tag encoding are assumed to belong to the length encoding without any kind of check. This implementation is not compliant with the X.690 DER encoding standard, it causes a wrong interpretation and the rejection of all certificates using multi-part tags.

```
...
Utils_MemSet(token, 0, sizeof(*token));
token->class = (((*lexer->current & 3<<6)) >> 6);
token->is_primitive = (!((*lexer->current & 1<<5) >> 5));
token->type = (*lexer->current & 31);
lexer->current++;
if ((*lexer->current & (1<<7)) != 0) {
    unsigned int num_octets = (*lexer->current & ((1<<7)-1))
    ;
    token->length = 0;
    for(unsigned int n = 0; n < num_octets; n++) {
        lexer->current++;
        token->length = (token->length << 8) | (*lexer->
    current);
    }
} else {
    token->length = *lexer->current & ((1<<7)-1);
}
lexer->current++;
...
```

Listing 9.12. Function for parsing tag and length of an element: the multi-part tag recognition and parsing is not implemented.

The main design difference of the second version of the certificate parser with respect to the first one consists in the error management: while the first version quits the whole procedure when an error condition is detected, the second one performs a bitwise OR of the return values of all the sub-procedures, which take value 0 if no errors occur and positive values as error codes. The purpose is to make the execution time as constant as possible, and thus to harden timing-based side-channel attacks whose aim is to leak implementation details. This design choice is prone to the buffer overflow memory errors described afterwards, as each sub-procedure does not repeat all the checks that were already performed by sub-procedures previously executed.

4. The *X509CertParser_ParseSerial()* function (Listing 9.13) parses the certificate serial number, copying it into the dedicated field of the certificate structure. In line 7, it accesses the first value octet of the current element before checking its length. If this element is empty and it is the last one in the certificate binary buffer, then *data* points to the first byte outside the buffer and dereferencing it causes a reading buffer overflow.

```
1  static X509CertParser_ErrorCode_t X509CertParser_ParseSerial(
2          X509CertParser_Certificate_t *cert,
3          X509CertParser_Parser_t *parser) {
4      const X509CertParser_Element_t *element = parser->element;
5      X509CertParser_ErrorCode_t ret_val = X509_PARSER_OK;
6      ret_val |= NOT(IS_INTEGER(element));
7      if ((element->data[0] == NULL_U8) && (element->length == VAL_U32(9))) {
8          Utils_MemCpy(cert->serial, &(element->data[1]), VAL_U32(8));
9      } else if ((element->data[0] != NULL_U8) && (element->length == VAL_U32(8))) {
10         Utils_MemCpy(cert->serial, element->data, VAL_U32(8));
11     } else {
12         ret_val = X509_PARSER_NOK;
13     }
14     return (ret_val);
15 }
```

Listing 9.13. Function for parsing the serial number: the value of the current element is accessed without checking the length.

The certificate shown in Listing 9.14 is not well-formed, since the first element should be a sequence containing the two main subsections, i.e. to-be-signed and signature.

```
1  unsigned char certBuf[0x8] = {
2          0x60,   // T: application specific, constructed;
```

```
    0x06,   // L: definite short, 6 octets value;
        0x60,   // T: application specific, constructed;
        0x04,   // L: definite short, 4 octets value;
            0x20,   // T: universal, constructed;
            0x02,   // L: definite short, 2 octets value;
                0x60,   // T: application specific, constructed;
                0x00    // L: definite short, 0 octets value;
};
```

Listing 9.14. Certificate triggering Vulnerability 4: the first element should be a sequence and the fourth one is empty.

The main procedure detects this and keeps trace of the error condition, but all the subsequent sub-procedures are executed anyway. The fourth element is interpreted as the container of the certificate serial number: since its length is 0, when *X509CertParser_ParseSerial()* tries to access the first value octet it triggers the aforementioned memory error.

5. The *X509CertParser_ParseVersion()* function (Listing 9.15) parses the certificate version number from the current element, after having verified that its tag encoding is context-specific and of invalid type. In line 10, it dereferences a pointer to the first value octet, without priorly performing any check on its length. Similarly to Vulnerability 4, if this pointer points to the first memory cell after the end of the certificate buffer above, dereferencing it causes a buffer overflow. Furthermore, the conditional statement in line 9 does not check if the value encoding of the fourth element is primitive or constructed, thus not preventing the execution of the true branch in the latter case. Another bug, in line 10, this time visible from manual code analysis, is that the integer value stating the certificate version is directly assigned to the corresponding field into the parsed certificate structure: this field is an enumeration with a set of 3 allowed values, but it is actually allowed to take any integer value. However this causes no problems to the subsequent phases of the authentication procedure, as the certificate version is not used.

```
1 static X509CertParser_ErrorCode_t X509CertParser_ParseVersion(
2         X509CertParser_Certificate_t *cert,
3         X509CertParser_Parser_t *parser) {
4     const X509CertParser_Element_t *element = parser->element;
5     X509CertParser_ErrorCode_t ret_val = X509_PARSER_OK;
6     if (X509CertParser_Is(element, TAG_CLASS_CONTEXT, TAG_TYPE_INVALID) == TRUE) {
        ret_val |= X509CertParser_Descend(parser);
        ret_val |= X509CertParser_NextElement(parser);
        if (IS_INTEGER(element) == TRUE) {
            cert->version = *element->data;
        } else {
            ret_val |= X509_PARSER_NOK;
        }
    }
    ret_val |= X509CertParser_Ascend(parser, VAL_U8(1));
    return (ret_val);
}
```

Listing 9.15. Function for parsing the certificate version: the content of the fourth element is accessed without checking the length, the certificate version can take any value.

The certificate shown in Listing 9.16 is not well-formed for three reasons: the first element should be a sequence, the tag encoding of the fourth element states that its value is at the same time universal, primitive and integer, which is not an allowed native tag, and the fourth element, that specifies the certificate version number, has no value.

```
unsigned char certBuf[0x8] = {
        0x60,   // T: application specific, constructed;
        0x06,   // L: definite short, 6 octets value;
            0x60,   // T: application specific, constructed;
            0x04,   // L: definite short, 4 octets value;
                0xa0,   // T: context specific, constructed;
                0x02,   // L: definite short, 2 octets value
    ;
                    0x22,   // T: universal, constructed,
    integer;
                    0x00    // L: definite short, 0 octets
    value;
};
```

Listing 9.16. Certificate triggering Vulnerability 5: the first element should be a sequence, the tag encoding of the fourth element is not allowed, the certificate version has no value.

The parser detects and keeps trace of the first error condition, like with the certificate in Listing 9.14. When *X509CertParser_ParseVersion()* parses the fourth element, it does not recognize the second and third error conditions, causing a memory error because of the latter.

6. The *X509CertParser_NextElement()* function (Listing 9.17) isolates an element beginning from the octet pointed by the current pointer to the certificate buffer, it extracts its tag and value length, and it deducts its depth in the

hierarchy of the elements.

```
X509CertParser_ErrorCode_t X509CertParser_NextElement(
        X509CertParser_Parser_t *parser) {
    X509CertParser_Element_t *element = parser->element;
    ...
    parser->current++;
    if (CONTENT_IS_LONGFORM(*parser->current) == TRUE) {
        unsigned int num_octets = CONTENT_NUM_OF_OCTETS(*parser->current);
        element->length = NULL_U32;
        for(unsigned int n = NULL_U32; n < num_octets; n++) {
            parser->current++;
            element->length = (element->length << VAL_U32(8)) | (*parser->current);
        }
    } else {
        element->length = *parser->current & CONTENT_LENGTH_MASK;
    }
    ...
```

Listing 9.17. Function for isolating an element and parsing its type and length: the length encoding is accessed without performing any check on the bounds of the certificate binary.

The main difference in the certificate shown in Listing 9.18 with respect to the the ones in Listing 9.14 and in Listing 9.16 is the incoherent length encoding: the value of the first element is correctly stated to be 6 octets long, while the value lengths of the second and third elements are both wrong by one unit, and the value length of the fourth element is in long form, which requires the presence of another subsequent octet stating the actual value length. Furthermore, since long form is used, the value length must be at least 128 to comply with the DER specifications, that prescribe to use the shortest possible length encoding: this is another reason of incompatibility between the lengths of the fourth elements and of the other elements. The parser does not detect the aforementioned incoherence, but, as usual, only that the first element is not a sequence.

```
unsigned char certBuf[0x8] = {
        0x60,   // T: application specific, constructed;
        0x06,   // L: definite short, 6 octets value;
            0x60,   // T: application specific, constructed;
            0x03,   // L: definite short, 3 octets value;
                0x20,   // T: universal, constructed;
                0x01,   // L: definite short, 1 octet value;
```

```
                        0x00,    // T: universal, primitive;
                        0x81     // L: definite long, 1 octet length;
};
```

Listing 9.18. Certificate triggering Vulnerability 6: wrong length encodings in the second, third and fourth elements.

The fourth time *X509CertParser_NextElement()* is called, it tries to parse the fourth element: since its length encoding is in long form, the true branch of the conditional statement is executed; during the first iteration of the for cycle, in line 11, it tries to access the first octet of the actual value length encoding, which does not exist, and instead it dereferences a memory cell outside the certificate buffer, causing a buffer overflow memory error. Since this function has access to the parent element through the *parser* structure, it is possible to prevent this out-of-bounds access by checking against the already parsed length of its parent and deducing that the for loop must not be executed.

7. In the certificate shown in Listing 9.19, contrarily to all the previous ones, the third element is primitive, thus it directly encodes a value, and the fourth element is a child of the second one, not of the third. Furthermore, the fourth element is not well-formed as the length encoding, which should always be present even if equal to 0, is missing. The third element is not context-specific, which means that *X509CertParser_ParseVersion()* does not interpret it as the container for the fourth element (line 6), which should be an integer specifying the certificate version, and thus *X509CertParser_NextElement()* is called 4 times.

```
unsigned char certBuf[0x8] = {
        0x60,    // T: application specific, constructed;
        0x06,    // L: definite short, 6 octets value;
            0x60,    // T: application specific, constructed;
            0x04,    // L: definite short, 4 octets value;
                0x00,    // T: universal, primitive;
                0x01,    // L: definite short, 1 octet value;
                    0x00,
                0x00     // T: universal, primitive;
};
```

Listing 9.19. Certificate triggering Vulnerability 7: wrong length encodings in the fourth element.

The fourth time *X509CertParser_NextElement()* is called, it tries to parse the fourth element: in line 6 of the function above, it tries to access the first octet of the length encoding in order to determine if it is in long or short form; since this octet does not exist, the current pointer actually points to a memory call outside the certificate buffer, triggering a buffer overflow memory error. This situation can be avoided considering the value length of the parent element, equal to 4, which allows to deduce that the last child element is not well-formed.

8. Another portion of the *X509CertParser_NextElement()* function is reported in Listing 9.20, that parses the tag encoding.

```c
X509CertParser_ErrorCode_t X509CertParser_NextElement(
        X509CertParser_Parser_t *parser) {
    X509CertParser_Element_t *element = parser->element;
    ...
    Utils_MemSet((unsigned char *) element, NULL_U8, sizeof(*element));
    element->class        = TAG_CLASS(*parser->current);
    element->is_primitive = TAG_IS_PRIMITIVE(*parser->current);
    element->type         = TAG_TYPE(*parser->current);
    parser->current++;
    ...
```

Listing 9.20. Function for isolating an element and parsing its type and length: the tag encoding is accessed without performing any check on the bounds of the certificate binary.

The certificate shown in Listing 9.21 is not well-formed because the first two elements should be sequences and the fourth one should be an integer, containing the certificate version. Differently from the certificates in Listing 9.18 and in Listing 9.19, the third element is context-specific, which means that *X509CertParser_ParseVersion()* takes the true branch of the conditional statement in line 6, thus *X509CertParser_NextElement()* is called 5 times instead of 4.

```c
unsigned char certBuf[0x8] = {
        0x60,   // T: application specific, constructed;
        0x06,   // L: definite short, 6 octets value;
            0x60,   // T: application specific, constructed;
            0x04,   // L: definite short, 4 octets value;
                0xa0,   // T: context specific, constructed;
                0x02,   // L: definite short, 2 octets value;
                    0x60,   // T: application specific, constructed;
                    0x00    // L: definite short, 0 octets value;
};
```

Listing 9.21. Certificate triggering Vulnerability 8: the first two elements should be sequences and the fourth one an integer.

The fifth time *X509CertParser_NextElement()* is executed, it tries to parse a non-existent fifth element: to determine the tag class (universal, application-specific or context-specific) the current pointer, pointing to the first octet of the tag encoding, needs to be dereferenced, but since it points to a memory cell outside the certificate buffer, it causes a buffer overflow memory error. Again, it is possible to avoid executing this portion of code by considering the value length of the parent element, i.e. the fourth, which is 0.

9. The certificate shown in Listing 9.22 has incoherent value lengths in all its elements: the first one should be 7, the second 5, the third 3, the fourth 1. *X509CertParser_NextElement()* detects these error conditions since the first time it is called to analyse the first element, but the procedure does not stop regardless. The third element is not context-specific, thus *X509CertParser-_ParseVersion()* deduces that the certificate version number is not specified and the fourth element is interpreted as the certificate serial number.

```
unsigned char certBuf[0x9] = {
        0x60,   // T: application specific, constructed;
        0x06,   // L: definite short, 6 octets value;
            0x60,   // T: application specific, constructed;
            0x03,   // L: definite short, 3 octets value;
                0x20,   // T: universal, constructed;
                0x01,   // L: definite short, 1 octet value;
                    0x60,   // T: application specific,
    constructed;
                    0x08,   // L: definite short, 8 octets
    value;
                        0x01    // T: universal, primitive,
    boolean;
};
```

Listing 9.22. Certificate triggering Vulnerability 9: all the value lengths are incoherent.

X509CertParser_ParseSerial() operates on the fourth element, whose value length must be 8 or 9. In this case, since the length is 8 and the value of the first octet is different from 0, the true branch of the second conditional statement is executed (line 10): the *Utils_MemCpy()* function tries to copy 8 octets of data from the certificate buffer to the parsed certificate structure. But all these octets are outside the certificate buffer and this causes a buffer overflow memory error. *X509CertParser_ParseSerial()*

could theoretically avoid this situation by reasoning on the elements' hierarchy, since it has access to the *parser* structure, but this is a responsibility of *X509CertParser_NextElement()*, that has already detected the error: *X509CertParser_ParseSerial()* trusts the parsed element structure filled by *X509CertParser_NextElement()*, but has no access to its error condition flag.

Since no more bugs were automatically triggered with small buffers (in the order of tens of bytes), the analysis stepped onwards with buffers that more closely resemble in size real certificates (in the order of a thousand bytes).

10. The *X509CertParser_UpdateDepth()* function (Listing 9.23) operates on the *parser* structure, which keeps the global status of the parser, and by considering the pointer to the first octet of the current element and an array containing the pointers to all its ancestors, it adjusts the depth level field inside *parser*. In order to index the ancestors array, the depth level field is used.

```
static void X509CertParser_UpdateDepth(
        X509CertParser_Parser_t *parser) {
    while ((parser->current == parser->parents[parser->depth
    ]) && (parser->depth > 1)) {
        parser->depth--;
    }
}
```

Listing 9.23. Function for adjusting the depth level counter in the structure containing the global status of the parser.

The certificate shown in Listing 9.24 is not well-formed for various reasons (e.g. wrong value length in the first element, the second element is not a sequence, ...), but its most important aspect is that the elements' hierarchy reaches a depth of 11 levels, while the maximum allowed is 10.

```
unsigned char certBuf[0x400] = {
        0x30,    // Level 0
        0x82,0x04,0x00,
            0x60,    // Level 1
            0x7f,
                0x60,    // Level 2
                0x7c,
                    0x60,    // Level 3
                    0x7a,
                        0x60,    // Level 4
                        0x77,
                            0x60,    // Level 5
                            0x75,
                                0x20,    // Level 6
                                0x68,
                                    ...
                                    0x20,   // Level 7
                                    0x00,
                                        // empty
                                    ...
                                    0xe0,   // Level 7
                                    0x01,
                                        0x00,
                                    0x20,   // Level 7
                                    0x06,
                                        0x63,   // Level 8
                                        0x04,
                                            0xa1,   // Level 9
                                            0x02,
                                                0x72,   // Level 10
                                                0x00,
                                                    // empty
                                    0x20,   // Level 7
                                    0x20,
                                    ...
};
```

Listing 9.24. Certificate triggering Vulnerability 10: the elements' hierarchy reaches a depth of 11 levels.

X509CertParser_NextElement() is capable of detecting such a situation and reporting an error condition, nonetheless *X509CertParser_UpdateDepth()* is executed and when the certificate above reaches level 10, it performs an out-of-bounds memory access, causing a buffer overflow. The described situation could be easily avoided by priorly checking that the depth value is in-bounds, or (more coherently with the design of the module) by controlling the error codes yielded by previous calls to *X509CertParser_NextElement()*.

Providing to the parser an unconstrained symbolic buffer with a size in the order of a thousand bytes proved to be a source of path explosion: such a simplistic test harness was not able to automatically trigger other bugs besides Vulnerability 10 in a reasonable amount of time, which was assumed to be in the order of tens of hours. The bugs described above affect functions called in the first steps of the parsing procedure, that parse fields which appear first in a X.509 certificate, such as the version and the serial. In order to attempt to trigger bugs in functions called in following phases of the parsing procedure, the adopted approach consisted in taking a well-formed certificate (with coherent DER encoding and with all the required fields) and make symbolic only specific parts of it. Since the type of errors that most likely were expected to be found were buffer overflows, only one element of the buffer at a time was made symbolic, specifically the last one, since if a pointer goes out of the bounds by one byte of an element in the middle of the buffer, it still points to a memory cell inside the buffer, thus it does not trigger a memory error.

11. The signature section of a X.509 certificate may optionally begin with a padding byte, which if present is always set to zero and must be removed in order to isolate the the signature value. The *X509CertParser_RemovePadding()* function (Listing 9.25) performs this task.

```
static void X509CertParser_RemovePadding(
        X509CertParser_Element_t *element_in,
        X509CertParser_Element_t *element_out) {
    if ((element_in->data[0] == NULL_U8) && ((element_in->
    length % VAL_U32(256)) == VAL_U32(1))) {
        element_out->data = &(element_in->data[1]);
        element_out->length = element_in->length - VAL_U32
    (1);
    } else {
        *element_out = *element_in;
    }
}
```

Listing 9.25. Function for removing the optional first padding byte.

A 953 bytes long certificate buffer was uses as a starting point for this part of the analysis. The first step consisted in making symbolic the last element, containing the whole signature section, but no bugs were triggered. The second step consisted in removing the signature section (made of 256 bytes of signature and 5 bytes of meta-data) and making symbolic one by one the elements containing the fields belonging to the signing algorithm section: making symbolic the container element of the signing algorithm section allowed to trigger a bug previously unreachable in a feasible time. The most significant parts of the bug-triggering certificate is shown in Listing 9.26: the symbolic execution engine reduced the size of the signing algorithm section by 2 bytes, making room for a signature section with no content.

```
unsigned char certBuf[692] = {
        0x30,   // Main container
        0x82,0x02,0xB0, // 688 bytes
            0x30,   // Container of to-be-signed section
            0x82,0x02,0x6D, // 621 bytes
            ...
            0x30,   // Container of signing algorithm
    section
            0x3B,   // 59 bytes
```

```
                ...
                0x03,    // Fake signature section
                0x00
};
```

Listing 9.26. Certificate triggering Vulnerability 11: a signature section with no content was automatically generated.

X509CertParser_RemovePadding(), in line 4, tries to access a non-existent first value octet of the element without priorly checking the value length, which was coherently set to 0 by *X509CertParser_NextElement()*, that filled the structure here received as first parameter. Since this functions tries to access a memory cell outside the certificate buffer, it causes a buffer overflow memory error. The simplest possible solution to prevent this out-of-bound access is to modify the condition of the if-then-else statement in line 4 so that the value length is checked before dereferencing the pointer to the value.

12. The parser checks that the container of the signing algorithm is identical to the container of the signed algorithm, which is a sub-section of the to-be-signed section. This task is carried out by the *X509CertParser_EquateElement()* function (Listing 9.27), that first checks the equivalence of the meta-data and then delegates the comparison of the value buffer to the *Utils_MemCmp()* function, whose implementation returns immediately with a false value if it encounters two different bytes before the end of the buffers specified by the length parameter.

```
unsigned char X509CertParser_EquateElement(
          const X509CertParser_Element_t *a,
          const X509CertParser_Element_t *b) {
     return (a->length       == b->length)       &&
            (a->class        == b->class)        &&
            (a->type         == b->type)         &&
            (a->is_primitive == b->is_primitive) &&
            (Utils_MemCmp((const unsigned char *) a->data, (
    const unsigned char *) b->data, a->length) == TRUE);
}
```

Listing 9.27. Function for comparing two elements, here used for the signing algorithm section.

In the certificate shown in Listing 9.26 the length encoding in the parents of the removed elements were adjusted: since the signature section is 261 bytes long, the length encoding of the main container was reduced from 949 to 688 bytes. The purpose was to trigger deeper bugs in the code: if a length encoding is wrong and/or incoherent with other length encodings, most likely this would trigger already discovered bugs in functions that isolate elements and parse their tag and length, rather than in functions used do parse specific fields, executed later. However, omitting these adjustments can simulate the effect of an incompletely received valid certificate, which may be interesting as well. The certificate shown in Listing 9.28 has been obtained by removing the signature section and the last two elements of the signing algorithm section from the original certificate, without adjusting any of the length encodings of the parent elements, but adjusting the size of the buffer in order to maximize the probability of finding buffer overflows related to the parsing of the last element's content. Then, the whole signing algorithm section was made symbolic.

```
unsigned char certBuf[953 - 261 - 5] = {
        0x30,    // Main container
        0x82,0x03,0xB5, // Not adjusted length
            0x30,    // Container of the to-be-signed section
            0x82,0x02,0x6D, // Not adjusted length
            ...
            0x30,    // Container of the signing algorithm
    section
            0x3D,    // Not adjusted length
                0x06,
                0x09,
                ...
                0x30,
                0x30,    // Not adjusted length
                    0xA0,
                    0x0D,
                    ...
                    0xA1,
                    0x1A,
                    ...
};
//                  0xA2,    // Element removed from the
    signing algorithm section
//                  0x03,
//                      0x02,
//                      0x01,
//                          0x20
```

Listing 9.28. Certificate triggering Vulnerability 12: the signing algorithm section is incomplete and the lengths were not adjusted accordingly.

Utils_MemCmp() receives as first parameter an element with a wrong length, 5 bytes more than the ones actually contained in the buffer starting from the signing algorithm section, therefore when trying to access the first one of these it causes a buffer overflow. The signing algorithm container element created by the symbolic execution engine in order to trigger this bug is identical to the signed algorithm element, in order to avoid that *Utils_MemCmp()* returns before having reached the first byte outside the buffer. Both *X509CertParser-_EquateElement()* and *Utils_MemCmp()* have to assume that the elements received as parameters are correct. A possible solution to avoid such a situation is to add a parameter to *X509CertParser_EquateElement()* that allows to observe the return status of *X509CertParser_NextElement()*, that is capable of identifying and reporting this error, in order to decide whether or not to invoke *Utils_MemCmp()*.

13. For a X.509 certificate to be considered valid it is necessary that the current time-stamp is in between the valid-not-before and valid-not-after time-stamps. The corresponding elements have both 0x17 as tag number, which states that the value is encoded into the UTC Time ASN.1 native type: the *X509CertParser_DecodeTime()* function (Listing 9.29) decodes them into Unix time-stamp values, which are the seconds elapsed since January 01, 1970. First, it isolates the sub-fields of the element value, i.e. year, month, day, hour, minute, second; then, it checks that all the sub-fields are in-bounds; finally, all the sub-fields are converted into second and summed together.

```c
X509CertParser_ErrorCode_t X509CertParser_DecodeTime(
        const X509CertParser_Element_t *element,
        X509CertParser_Time_t          *time) {
    static const unsigned char daysPerMonth[12] =
    {/*Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec*/
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    ...
    if ((month < 1) || (month > 12)) {
        ret_val |= PARSER_NOK;
    }
    if (day < 1) {
        ret_val |= PARSER_NOK;
    } else if ((is_leap) && (month == 2) && (day > 29)) {
        ret_val |= PARSER_NOK;
    } else if (day > daysPerMonth[month - 1]) {
        ret_val |= PARSER_NOK;
    }
    ...
    year  -= 1970;
    month -= 1;
    day   -= 1;
    *time = year * SECONDS_PER_YEAR;
    for (i = 0; i < month; i++) {
        *time += daysPerMonth[i] * SECONDS_PER_DAY;
    }
    *time += day   * SECONDS_PER_DAY;
    *time += hour * SECONDS_PER_HOUR;
    *time += minute * SECONDS_PER_MINUTE;
    *time += second;
    *time += leap_days * SECONDS_PER_DAY;
    return (ret_val);
}
```

Listing 9.29. Function for decoding an UTC Time ASN.1 native type into a Unix Timestamp.

The certificate buffer shown in Listing 9.30 triggers a bug in *X509CertParser-_DecodeTime()*: all the elements that follow the valid-not-before and valid-not-after sections were removed, the length encodings and the size of the buffer were adjusted coherently and the value of the valid-not-after element was made symbolic (not so the tag and length encodings). The obtained month and day values are out-of-range and the end-of-string value appears two octets before than in the valid-not-before element, thus the seconds are not specified.

```
unsigned char certBuf[953
    -261     // Signature
    -63 // Signing algorithm
    -6 -7 -4 -7 -73 -11 -275-2-1-1 -2 -15 -13 -18 -23] = {
        0x30,0x81,0xA8, // Main container
            0x30,0x81,0xA5, // Container of to-be-signed
                ...
                0x30,0x1E,  // Validity section
                    0x17,0x0D,  // Valid-not-before section
                        0x31,0x37,  // Year = (20)17
                        0x30,0x34,  // Month = 4
                        0x30,0x37,  // Day = 7
                        0x31,0x31,  // Hour = 11
                        0x34,0x37,  // Minute = 47
                        0x30,0x30,
                        0x5A,       // End of UTC Time
   string
                    0x17,0x0D,  // Valid-not-after section
                        0x31,0x30,  // Year = (20)10
                        0x32,0x30,  // Month = 20
                        0x30,0x30,  // Day = 0
                        0x30,0x30,  // Hour = 0
                        0x30,0x30,  // Minute = 0
                        0x5A,       // End of UTC Time
   string
                        0x00,0x00
};
```

Listing 9.30. Certificate triggering Vulnerability 13: the month and day values of the valid-not-after-section are out-of-range.

X509CertParser_DecodeTime(), in line 8, verifies that the month value is in-bounds: if it is not, this error condition is reported but the function does not return. Then in the for loop starting from line 23, this value is used to index the *daysPerMonth* array, in order to add to the time-stamp the correct amount of seconds. In the certificate in Listing 9.30 the month value is 20, therefore when the index variable takes the value 13, the function tries to perform an out-of-bounds access, triggering a buffer overflow. This error can be avoided simply by extending the condition of the for loop with a check on the index, that should not be allowed to take a value greater than 11.

14. A bug in *X509CertParser_DecodeTime()* analogous to Vulnerability 13 is triggered by the certificate shown in Listing 9.31, which has been obtained by setting the test harness exactly in the same way as for the certificate in Listing 9.30, the only difference consists in the sample value for the valid-not-after section, which now has a valid day value.

```
unsigned char certBuf[953 -261 -63 -6 -7 -4 -7 -73 -11 -275-2-1-1 -2 -15 -13 -18 -23] = {
        0x30,0x81,0xA8, // Main container
            0x30,0x81,0xA5, // Container of to-be-signed
                ...
                0x30,0x1E,  // Validity section
                    0x17,0x0D,  // Valid-not-before section
                        0x31,0x37,0x30,0x34,0x30,0x37,0x31,0x31,0x34,0x37,0x30,0x30,0x5A,
                    0x17,0x0D,  // Valid-not-after section
                        0x31,0x30,   // Year = (20)10
                        0x32,0x30,   // Month = 20
                        0x31,0x30,   // Day = 10
                        0x30,0x30,   // Hour = 0
                        0x30,0x30,   // Minute = 0
                        0x5A,        // End of UTC Time string
                        0x00,0x00
};
```

Listing 9.31. Certificate triggering Vulnerability 14: the month value of the valid-not-after-section is out-of-range.

The different value for the day makes the control flow follow an execution path along the condition of the if-then-else statement in line 15, which again uses the month value to index the *daysPerMonth* array, this time in order to determine if the specified day exists in the specified month. When the month takes a value above 12 (it has not been decremented by one in line 20 yet) it triggers a buffer overflow memory error.

9.5 Certificate Manager - X.509 certificate validation

The analysis of the Certificate Manager can be split in two parts: X.509 certificate validation procedure and cryptographic challenge validation procedure. For the former, the main function of the test harness instantiates an unconstrained symbolic buffer with fixed size, intended to be modified manually, as shown in Listing 9.32.

```
#define CERT_LEN
#define CERT_VALID_PHASE 0
unsigned char certBuf[CERT_LEN];

klee_make_symbolic(certBuf, CERT_LEN, "certBuf");

unsigned char result;
CertificateManager_SetCertificateContext(certBuf, CERTL_LEN, &
    result);
CertificateManager_Authenticate(CERT_VALID_PHASE);
...
```

Listing 9.32. Main function of the test harness for testing the Certificate Manager module.

The initially adopted approach was fundamentally the same already used in Section 9.2: to begin with buffers with size in the order of tens of bytes and to escalate to size that approximate the ones of real certificates. However, the obtained results were not meaningful, as they were equivalent to the ones reported in Section 9.3 and in Section 9.4: it was not possible in a reasonable time to explore new execution paths deriving from the logic of the Certificate Manager, and thus trigger new error conditions.

The test harness has been modified by adding a stub of the X.509 Certificate Parser, shown in Listing 9.33, which allows to focus the analysis solely on the certificate manager.

```
X509CertParser_ErrorCode_t X509CertParser_Stub_Parse(
        X509CertParser_Certificate_t *cert,
        const unsigned char *data,
        unsigned int num) {
    if (klee_is_symbolic(num)) {
        cert->issuer.data = data;
        cert->issuer.length = num;
        return klee_int("X509CertParser_Stub_Parse_Return");
    }
    for (unsigned int i = 0; i < num; i++) {
        if (klee_is_symbolic(data[i])) {
            cert->issuer.data = data;
            cert->issuer.length = num;
            return klee_int("X509CertParser_Stub_Parse_Return");
        }
    }
    return X509CertParser_Parse(cert, data, num);
}
```

Listing 9.33. Stub of the X.509 Certificate Parser used in the test harness for the Certificate Manager.

The Certificate Manager verifies that the issuer of the received X.509 certificate is equal to the subject of at least one of its statically configured certificates, then it uses the public key of the latter to verify the signature of the former. Therefore, the only field of the parsed certificate that it uses internally is the issuer: the to-be-signed section and the signature field are passed as input parameters to the Cryptographic Manager, which has been substituted with a stub and makes no use of any input parameter. The parser is used by the Certificate Manager both for the statically configured certificates and for the received certificate, which is symbolic. The stub above checks if the length of the buffer or at least one of its bytes are symbolic: if they are, then the buffer and its length are assigned to the issuer field of the "parsed" certificate, else the true parsing function is invoked with the same input parameters. This allows to have an unchanged behaviour for the statically configured certificates, to avoid parsing the (partially) symbolic input certificate (thus avoiding the related path explosion problem), and to inject symbolic values exactly where needed. Such an analysis is under-constrained and thus prone to false positives, which must be discarded: when a bug-triggering input is found, the parser stub must be substituted with the true one and the procedure re-run with that concrete input, in order to verify that it is still capable of triggering the same bug despite the constraints imposed by the true parser.

Describing these choices with the terminology of S2E, the parser becomes part of the environment together with the driver for the hardware cryptographic accelerator while only the Certificate Manager remains part of the unit, and the test harness implements again a form of Local or Over-approximate execution consistency model, as also in this case there are no constraints imposed by interface contract and creation of symbolic value is done in the environment.

15. The analysis of the certificate validation procedure yielded a single bug-triggering

buffer, consisting in exactly the same sequence of bytes of one among the statically configured certificates, beginning from its subject section. Actually the capability of the programming error to trigger the bug depends on the length of the input buffer, which is set statically and varied manually: it must be enough to contain the part of the static certificate that goes from the beginning of the subject to the last byte, plus one byte. To explain how the symbolic execution engine generated it, it is necessary to examine the core of the certificate validation procedure, reported in Listing 9.34.

```c
unsigned char idx = NULL_U8, valid_sts, *res;
const unsigned char *data;
unsigned int size;
X509CertParser_Certificate_t in_cert, auth_cert;
X509CertParser_ErrorCode_t pa_err;
...
pa_err = X509CertParser_Parse(&in_cert, data, size);
if (pa_err == X509_PARSER_OK) {
    valid_sts = TRUE;
    for (idx = NULL_U8; (idx < STATIC_CERTIFICATES_BINARIES_NUM) && (valid_sts == TRUE); idx++) {
        X509CertParser_Parse(
                    &auth_cert,
                    STATIC_CERTIFICATES_BINARIES[idx].cert,
                    STATIC_CERTIFICATES_BINARIES[idx].length);
        if (Utils_MemCmp(
                    in_cert.issuer.data,
                    auth_cert.subject.data,
                    in_cert.issuer.length) == TRUE) {
            valid_sts = CryptographicManager_VerifyToBeSigned(
                    auth_cert.publicKeyMod.data, auth_cert.publicKeyMod.length,
                    auth_cert.publicKeyExp.data, auth_cert.publicKeyExp.length,
                    in_cert.toBeSigned.data, in_cert.toBeSigned.length,
                    in_cert.signature.data, in_cert.signature.length);
            *res = valid_sts;
        }
    }
    ...
```

Listing 9.34. Core of the certificate validation procedure, inside the Certificate Manager.

The equivalence between issuer and subject is checked in the line 12 relying on the same *Utils_MemCmp()* function (Listing 9.35) that was involved in Vulnerability 12 and inside which also this bug is triggered.

```c
1  unsigned char Utils_MemCmp(
2          const unsigned char *ptr_1,
3          const unsigned char *ptr_2,
4          unsigned int size) {
5      unsigned char is_equal = FALSE;
6      if ((ptr_1 != NULLP_U8) && (ptr_2 != NULLP_U8)) {
7          is_equal = TRUE;
8          for (unsigned int i = NULL_U32; (i < size) && (
    is_equal == TRUE); i++) {
9              is_equal = (ptr_1[i] == ptr_2[i]);
10         }
11     }
12     return (is_equal);
13 }
```

Listing 9.35. Function for comparing two buffers: as soon as a mismatch is found, the function quits.

Contrarily to that previous situation, the root cause of the bug is not a mismatch between the actual length and the parsed length of the issuer buffer: the parsing of a static certificate is assumed to always succeed, while the exit status of the parsing of a received certificate is always checked, therefore the parsing must succeed for *Utils_MemCmp()* to be called and the two lengths are always coherent. *Utils_MemCmp()* is called using as length the one of the issuer of the received certificate, without any check of equality with the subject length: the former is thus completely unbounded and under the control of the certificate sender. As long as the issuer value is equal to the subject value and the following fields (including their tag and length encoding) the for loop in line 8 does not terminate and a long enough sequence of equal bytes causes an out-of-bounds access on the first array in line 9. Such an access may or may not trigger a buffer overflow memory error, depending on the architecture of the (virtual) machine on which this code is executed. The former case occurs when executing it inside the KLEE environment: trying to access the first byte after the end of one of the statically configured certificates triggers a memory error, as seen with all the previously reported bugs. This requires the symbolic execution engine to generate a sequence of bytes equal to the part of a static certificate beginning from the subject field. The latter case is likely to happen in real platforms, where going out-of-bounds while accessing an array does not necessarily trigger for example a segmentation fault, and it is a potentially more critical situation, since it leaves room for a timing based side-channel attack. Since the execution time of *Utils_MemCmp()* is dependent on the content of the issuer field, if an attacker is able to accurately measure it, it can deduct (part of) the content of the main memory. If the attacker guesses the first byte but not the second, then the for loop in line 8 performs two iterations, if it guesses the first and the second bytes but not the third, there will be three iterations, and so on. The attacker needs to try a maximum of 256 values for each single byte that it is willing to guess. This is not a serious problem for parts of a X.509 certificate that follow the subject field, as they are public by definition, but it is possible to discover the content of the main memory that follows a static certificate, the only limits being the maximum allowed length for an input certificate, if present, and the CPU and operating system's architecture, if they implement segmentation faults or analogous behaviours. This may lead to leakage of industrial secrets. The most obvious countermeasure for this bug is to check that the issuer and subject lengths are equal before calling *Utils_MemCmp()*.

As already mentioned, in order to verify that this bug can be triggered also in the original system, it is necessary to substitute the under-constraining stub implementation of the parser with the original one. It is not necessary to do the same with the stub for the cryptographic accelerator driver, as this memory error occurs before any of its functions are used. For the parser to return with a success exit status, the issuer of a valid certificate was replaced with the bug-triggering issuer found, as shown in Listing 9.36.

```
unsigned char certBuf[1682] = {
        0x30,   // Main container
        //0x82,0x03,0xB5,
        0x82,0x06,0x8E, // Adjusted length
            0x30,   // To be signed section
        //  0x82,0x02,0x6D,
            0x82,0x05,0x46, // Adjusted length
                ... // Content preceding issuer field
                0x30,
//              0x34,
                0x82,0x03,0x0B,
//                  0x31,   // Beginning of original issuer
//                  0x13,
//                      0x30,
//                      0x11,
//                          0x06,
//                          0x03,
//                              0x55,0x04,0x03,
//                          0x13,
//                          0x0A,
//                                  0x43,0x65,0x72,0x74,0x4C,0
    x65,0x76,0x65,0x6C,0x33,
                                ... // Remaining content of original
     issuer
                        0x31,    // Beginning of bug-triggering
    issuer
                        0x13,0x30,0x11,0x06,0x03,0x55,0x04,0x03,
                        0x13,0x0A,0x43,0x65,0x72,0x74,0x4C,0x65,
                        0x76,0x65,0x6C,0x34,0x31,0x10,0x30,0x0E,
                        0x06,0x03,0x55,0x04,0x0A,0x13,0x07,0x43,
                        ... // Remaining content of bug-
    triggering issuer
                ... // Remaining content of original certificate
};
```

Listing 9.36. Certificate triggering Vulnerability 15: the issuer is equal to one among the statically configured certificates, beginning from the subject section.

It is possible to notice that the issuer field is actually composed by multiple sub-fields, and that the bug-triggering issuer is not well-formed "internally". This should be noticed by the parser that as a consequence should return notifying the error condition, however, the parsing of the issuer consists solely in isolating the value of its container element: *Utils_MemCmp()* compares the binary encodings of issuer and subject, not their parsed sub-fields. Therefore, the bug described above can occur also in the original system. Given the inability to understand the behaviour of the Diagnostic Protocol module, not even with the help of symbolic execution, it has not been possible to verify the actual capability of such a malformed certificate to arrive to the Certificate Manager module starting from the CAN Interface module, where it is injected split across multiple CAN messages. In particular it has been impossible to deduce its maximum length: an ISO-TP payload can be up to 4095 bytes long, which may impose an upper bound on the maximum certificate length, but it is as well possible that the Diagnostic Protocol Module is capable of reassembling multiple ISO-TP payloads into a single certificate.

9.6 Certificate Manager - Cryptographic challenge validation

The analysis of the cryptographic challenge validation procedure is not meaningful. To explain the motivation, the core of this procedure is reported in Listing 9.37.

```
X509CertParser_Certificate_t auth_cert;
unsigned char idx, valid_sts, *p_res, cert_lvl;
const unsigned char *encrypted_challenge_data, *challenge_data,
    *rand_num_data;
```

```
 4 unsigned int encrypted_challenge_size, challenge_size,
     rand_num_size;
 5 CertificateManager_ErrorCode_t ret_val;
 6 ...
 7 x509_Stub_parse(
 8         &auth_cert,
 9         STATIC_CERTIFICATES_BINARIES[cert_lvl].cert,
10         STATIC_CERTIFICATES_BINARIES[cert_lvl].length);
11 ret_val = CryptographicManager_RsaDecrypt(
12         auth_cert.publicKeyExp.data, auth_cert.publicKeyExp.
     length,
13         auth_cert.publicKeyMod.data, auth_cert.publicKeyMod.
     length, FALSE,
14         encrypted_challenge_data, encrypted_challenge_size,
15         challenge_data, &challenge_size, FALSE);
16 if (ret_val == CLIB_NO_ERR) {
17     if (Utils_MemCmp(challange_data, rand_num_data,
     rand_num_size) == TRUE) {
18         *p_res = TRUE;
19     }
20 }
```

Listing 9.37. Core of the cryptographic challenge validation procedure, inside the Certificate Manager.

Apparently there is a programming error in line 10 of the same nature of the one in the X.509 certificate validation procedure: the decrypted and the original random numbers are checked for equality with *Utils_MemCmp()* using as length parameter the one of the original random number, which is not dependent on the user-provided inputs; the length of the buffer containing the decrypted random number may be smaller, and this would cause a buffer-overflow memory error, though less exploitable than Vulnerability 15. However, the length of the decrypted buffer is set by the driver for the hardware cryptographic accelerator, which has been substituted by a stub. It is possible to make the stub return a symbolic length value that would trigger this bug, but since the real hardware has not been available for this analysis, it has been impossible to determine if it could actually return such a length.

Chapter 10

Conclusions

In this work, the applicability of techniques of dynamic code analysis in the testing phase of a critical software system has been verified successfully. In this way, the amount of manual effort required by the tester in a traditional approach to white-box testing is effectively reduced. It is no more needed to thoroughly inspect and to precisely understand the inner working of the piece of software under test, in order to reason out an exhaustive set of test cases and all the possible information flows involving sensitive and/or not trusted data. The most promising among these techniques, symbolic execution and dynamic taint analysis, were applied on a real embedded system. The selected use case consists in a central gateway ECU operating in-car, providing two main functionalities: CAN messages filtering/routing and external diagnostic devices authentication.

The former has been analysed by means of DTA based on SE, considering the input channels as non trusted sources of data, and the output channels as destinations. If the SE engine generates a well-formed message, then all the output channels must remain intact, except the one to which the message should be forwarded. Data derived from a non trusted source must not be able to propagate to them. Otherwise, if a bad message was injected, all channel must preserve their integrity property.

The latter requires the unit to parse a public key certificate sent from the diagnostic device and to verify that it was emitted by one of the trusted certification authorities. These portions of the procedure naturally lent themselves to an analysis based on SE, because their control flow is heavily influenced by external data potentially under the control of an attacker. It has not been possible to analyse the part of the procedure performing cryptographic computations, because they generate constraints sets not solvable in a feasible time.

The necessity to deal with the limitations of SE in the second part of the analysis has lead to the definition of a template procedure for creating a test harness and focusing the analysis on the portions of code of interest.

The analysis of the messages filtering/routing functionality did not reveal any incongruence between the implementation and the available specifications listing all the admitted messages for each input channel. Since the analysis has been exhaustive, it is possible to state that its correctness has been proved. Furthermore, the results obtained with a tool implementing the traditional approach to DTA were compared with the ones obtained with the developed prototype. The former were not precise, since they were affected by false negatives. The latter were exact, since simplifying assumptions to deal with path explosion were not needed.

On the other side, the analysis of the authentication procedure revealed defects in the parser that did not emerge with a more traditional approach to the testing phase. All of them cause buffer overflow memory errors and most of them are caused by a single design flaw. Since the hardware platform on which this firmware was deployed was not available, it is unknown if these malfunction can appear during normal operation. The verification of the certificate's emitter is affected by a bug of the same type. Differently from all the others, this one is theoretically exploitable by means of a timing-based side channel attack, which allows an attacker to read the content of the main memory. This may cause leakage of sensitive information about the system.

A secondary contribution of this work has been the implementation of a recent algorithm for DTA based on SE. The sources of false negatives and false positives affecting the results can now more easily be controlled and, under certain circumstances, eliminated. The prototype was successfully applied in the analysis of the use case, and yielded exact results as opposed to a publicly available tool implementing the traditional approach.

Future developments based on this work include:

- Further verification of the scalability of the described techniques by means of analysis of larger pieces of software. Specifically, the firmware analysed in this work is part of a larger system composed of many others. The largest one among them has roughly 10 times the amount of lines of code, which makes it a suitable candidate.

- Extension of the analysis to the binary code to be deployed of the firmware. By testing the binary code, which includes the operating system and its drivers, it is possible to observe the actual failures caused by the defects found. However, since the binary code is less syntactically rich than the corresponding binary code, identification of defects like memory manipulation errors is harder. Furthermore, a greater effort for building a test harness is required. It is necessary to emulate the hardware of the platform and to find suitable injection points for symbolic values in the virtual peripherals. The S2E tool offers support for this kind of analysis, since it is based on QEMU, which provides an emulated ARM environment. This platform is similar to the one on which the considered use case runs and it must be properly adapted by adding the needed custom assembly instructions and custom embedded peripherals.

Bibliography

[1] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317–331, IEEE, 2010.

[2] C. Cadar and K. Sen, “Symbolic execution for software testing: three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[3] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[4] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.,” in *OSDI*, vol. 8, pp. 209–224, 2008.

[5] R. Corin and F. A. Manzano, “Taint analysis of security code in the klee symbolic execution engine,” in *International Conference on Information and Communications Security*, pp. 264–275, Springer, 2012.

[6] J. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 196–206, ACM, 2007.

[7] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: dynamic taint analysis with targeted control-flow propagation.,” in *NDSS*, 2011.

[8] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, “Verifying information flow properties of firmware using symbolic execution,” in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 337–342, EDA Consortium, 2016.

[9] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy*, 2016.

[10] R. Corin and F. A. Manzano, “PySymEmu - A symbolic execution tool, capable of automatically generating interesting inputs for x86/x64 binary programs.” https://github.com/feliam/pysymemu.

[11] S. McCamant, P. Saxena, D. Akhawe, *et al.*, “FuzzBALL.” https://github.

com/bitblaze-fuzzball/fuzzball.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* (W. Damm and H. Hermanns, eds.), vol. 4590 of *Lecture Notes in Computer Science*, pp. 519–531, Springer, 2007.

[13] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, pp. 1–25, Springer, 2008.

[14] W. M. Khoo, "Taintgrind: a Valgrind taint analysis tool." https://github.com/wmkhoo/taintgrind.

[15] C. Lattner and V. Adve, "LLVM language reference manual," 2006.

[16] ISO, "Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services," ISO 15765-2:2016, International Organization for Standardization, Geneva, Switzerland, 2016.

[17] ITU-T, "Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," ITU-T X.690, International Telecommunication Union, Geneva, Switzerland, 07 2002.

[18] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code.," in *USENIX Security Symposium*, pp. 49–64, 2015.

[19] H. Yoshida, G. Li, T. Kamiya, I. Ghosh, S. Rajan, S. Tokumoto, K. Munakata, and T. Uehara, "Klover: Automatic test generation for c and c programs, using symbolic execution," *IEEE Software*, vol. 34, no. 5, pp. 30–37, 2017.

[20] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *International Symposium on Engineering Secure Software and Systems*, pp. 58–72, Springer, 2011.