EPiC Series in Computing

Volume 87, 2022, Pages 1–17

Proceedings of 10th International Workshop
on Security Proofs for Embedded Systems

Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to Detect Fault Injection Vulnerabilities

Guilhem Lacombe[12], David Féliot[2], Etienne Boespflug[13], and Marie-Laure Potet[13]

[1] Université Grenoble Alpes, Grenoble, France
prenom.nom@univ-grenoble-alpes.fr
[2] CEA Leti CESTI, Grenoble, France
prenom.nom@cea.fr
[3] Verimag, Grenoble, France

Abstract

Certification through auditing allows to ensure that critical embedded systems are secure. This entails reviewing their cryptographic components and checking for dangerous execution paths. This latter task requires the use of specialized tools which allow to explore and replay executions but are also difficult to use effectively within the context of the audit, where time and knowledge of the code are limited. Fault analysis is especially tricky as the attacker may actively influence execution, rendering some common methods unusable and increasing the number of possible execution paths exponentially. In this work, we present a new method which mitigates these issues by reducing the number of fault injection points considered to only the most relevant ones relatively to some security properties. We use fast and robust static analysis to detect injection points and assert their impactfulness. A more precise dynamic/symbolic method is then employed to validate attack paths. This way the insight required to find attacks is reduced and dynamic methods can better scale to realistically sized programs. Our method is implemented into a toolchain based on Frama-C and KLEE and validated on WooKey, a case-study proposed by the National Cybersecurity Agency of France.

Keywords: fault injection robustness evaluation, source code static analysis, symbolic execution, WooKey bootloader use-case

1 Introduction

From our credit cards to medical equipment and methods of transport, embedded systems are relied upon in nearly all aspects of modern life, including critical and sensitive applications.

This work is partially supported by ANR-15-IDEX-02.

Trust in these devices and their protective mechanisms is therefore paramount to ensure the viability of our professional endeavors and lifestyles.

One way the security of such devices can be ascertained is through certification processes such as Common Criteria[17]. This allows to gauge the difficulty of finding and performing attacks on the target system by measuring the time and level of expertise required to do this for a group of approved auditors leveraging state-of-the-art equipment and techniques[10]. Included in their arsenal are perturbation attacks, also known as fault injection attacks, which aim to cause exploitable faulty behavior in a system by subjecting it to extreme operating conditions. Vectors of fault injection include applying strong electromagnetic fields to some components using a laser, causing power jolts or stressing DRAM memory via software to induce bitflips[23, 12]. This can lead to secret information such as bits of cryptographic keys leaking through side channels[28] or even to loading an outdated firmware[20, 31].

Some of the tasks auditors must fulfill include reviewing cryptographic components of the target program and exploring execution paths looking for dangerous behaviors. This latter task requires the use of tools which can have difficulties scaling to large programs due to path space explosion and infinite loops. Moreover these issues only get worse when considering an active attacker who may influence execution by injecting faults. Methods which could be used to reduce the analysis perimeter such as slicing are inadequate in this context as well. Considering that auditors have limited time and knowledge of the code, fault analysis tools tend to be impractical on realistically sized programs.

To mitigate these issues, we propose an approach using static analysis to detect the most relevant fault injection points in a program relatively to security properties which an opponent may want to attack. It involves finding faultable instructions in the dependencies of the properties and formally checking that they may have an impact. This way the number of execution paths to consider is reduced to only those which cannot be formally proven harmless. As a result more precise tools such as fault simulators, which generate mutated programs with simulated faults, and fault analysis tools based on dynamic symbolic execution[21, 16], which only generate a single mutated program for analysis, can be used on realistically sized programs more effectively.

In Section 2 of this paper we place our contributions within the current state of the art. We then present our method in Section 3 and provide experimental validation in Section 4. Finally, we discuss related works in Section 5.

2 Context and Contributions

As part of the certification process, the auditors working for the Information Technology Security Evaluation Facilities (ITSEFs) must identify potential vulnerabilities to fault attacks in their target. This *fault analysis* either helps to discover actual exploits or to assert that the target is secure against some attacker models. It also allows to find theoretical exploits which could not be performed within the limited time frame of the audit and would have been missed otherwise. When sources are available, auditors start from the source code and a state-of-the art in terms of attack scenarios, tracking logical vulnerabilities that can generally be replayed at the binary level[26].

2.1 Motivating Example

The following is a discussion of an example program which contains vulnerabilities to fault injection despite the presence of countermeasures. We will also use this example to illustrate our fault analysis method later.

```
 1 typedef struct data{
 2   size_t msg_size;                 //input > 255
 3   char msg[255];                   //input
 4   uint32_t key[8];                 //secret
 5 } data_t;
 6 
 7 void print_message(data_t *d){
 8   size_t size = d->msg_size & 0xff; //countermeasure, fault 0xff to 0
       xffffffff
 9   for(size_t i = 0; i < size; i++){
10     if(i >= size)                   //bypassed countermeasure
11       exit(1);
12     //@ assert \valid_read(&d->msg[i]); <- expected security property
13     printf(”%c”, d->msg[i]);        //bytes of the key in the output!
14   }
15   printf(”\n”);
16 }
```

Figure 1: Example of a function vulnerable to fault injection

The function presented on Figure 1 prints a message based on its size, both of which are controlled by the user. Under nominal circumstances this is not an issue since a mask is applied to the size on line 8, limiting its maximum effective value to 255 and thus preventing buffer overflows. The index is also checked to be within the expected bound of the buffer on line 10 in an attempt to thwart fault injection attacks.

However, attacking the countermeasure on line 8 by forcing the mask to 0xffffffff with a fault, which is a commonly considered outcome, results in the exact user provided size being used. This also bypasses the index check on line 10. In this example a secret cryptographic key is conveniently stored near the message in memory. Inputting a greater than 255 value as the message size will therefore result in bytes of the key leaking in a similar way as with the Heartbleed OpenSSL vulnerability[1] and violating the property expressed on line 12[1].

Ignoring the fact that storing a secret key in such a fashion is unadvisable, detecting such vulnerabilities to fault injection can be difficult when they are buried deep within an application. In fact, our example was inspired by an attack that was found on the ANSSI's WooKey project[2] in a library comprised of roughly 2.5k lines of code[2][3], which violated a similar property to the one on line 12, leading to a stack buffer overflow. Additionally, the presence of some commonly used countermeasures may hide the issue to visual inspection. The use of automated analysis tools is therefore required, not only in order to be able to reliably detect fault injection vulnerabilities, but also to evaluate the effectiveness of countermeasures. Unfortunately such tools tend to struggle with scaling to realistically sized programs because of path space explosion and infinite loops occurring as a result of faults. Users must therefore reduce the size of their target, which we refer to as the *analysis perimeter*, in order to use them.

[1] In ACSL, *valid_read* allows to check that the content of pointers can be safely read as per the C standard.

[2] See the 2020 Inter-CESTI challenge report[11], section 9.

[3] See the SC_get_ATR function.

```
 1 void process(int a, int b){ //function to analyze
 2   if(a && b){               //nominal path
 3     if(!a || !b) exit(1);   //countermeasure
 4     assert(a && b);
 5     ...
 6     return;
 7   }
 8   if(a){                    //reachable by faulting the previous test
 9     if(!a) exit(1);         //countermeasure
10     assert(a && !b);
11     ...
12     return;
13   }
14   ...
15 }
16
17 int analysis_main(){
18   process(1, 1);
19   return 0;
20 }
```

Figure 2: Program with an unfeasible execution path becoming feasible with a fault

2.2 Difficulties in defining an Analysis Perimeter for Fault Analysis

Extracting an analysis perimeter is usually done by expressing assertions related to security properties and slicing[30, 22] based on dependencies. This allows to produce a minimal program corresponding only to relevant execution paths with regard to assertions. The issue with this method is that faults may redirect control flow and induce paths which are normally unfeasible. It could therefore result in the loss of attack paths.

The example from Figure 2 shows a program setup for the analysis of the *process* function. In this case, both *a* and *b* are set to 1, which should result in the test on line 2 being always positive under nominal condition. However a fault can be used to invert the result of this test, which would result in the execution reaching the one on line 8. Since the countermeasure there does not check that *b* is null, the assertion line 10 would be violated. However this execution path would be lost after slicing since it would be detected as unfeasible by a precise dependency analysis, resulting in a potential attack being missed and illustrating the fact that this approach is not adapted for fault analysis. Note that slicing may also result in countermeasures being removed.

The analysis perimeter for fault analysis should therefore contain all control-flow paths in the control flow graph of the program, regardless of feasibility, in order to avoid loss of behavior. By definition, this means that the usefulness of slicing would be severely limited.

The consequence of these difficulties is that reducing the size of the analyzed program is often not possible in the context of fault analysis. There is however another way in which we may effectively reduce the analysis perimeter, which is to reduce the number of fault injection points considered.

2.3 Contributions

- We propose a method using static value analysis to find the injection points that a security property depends on and formally check that they may have an impact. Relevant injection points are then selected for further analysis with a dedicated fault analysis tool.

- We present an implementation of our method as a toolchain based on proven and widely used tools which is suitable for both single- and multi-fault analysis. The static analysis part is handled by Frama-C and some of its plugins. Our fault analysis tool of choice is Lazart, which is itself based on dynamic symbolic execution by KLEE.

- We validate our method by using our implementation to find fault injection vulnerabilities in the ANSSI's WooKey secure USB storage device[3]. This results in weaknesses being discovered in the countermeasures of the bootloader part. We also complement our experiments with an analysis of the *sudo* unix command[4].

Our method allows to reduce the number of fault injection points considered in the fault analysis of a program by removing those which can be formally proven to have no impact on security properties. The use of static analysis to this end allows to handle difficult execution paths and should result in better scalability and performance when eliminating false positives with dynamic symbolic execution, as illustrated in section 4.

3 Our Method

Our goal with this work is to design a method allowing to reduce the number of injection points considered for fault analysis without risking to remove important parts of the target program nor losing in precision in order to improve the scalability of other fault analysis methods. We will focus on source code analysis to better assist with code comprehension and evaluation. We will also implement our method as a tool-chain based on proven and widely used tools.

3.1 Tools

Frama-C: Frama-C[29] is a static analysis platform for the C language. It parses .c files into a formal AST structure (based on CIL) which accommodates annotations detailing specifications and properties using the ACSL specification language[5]. Frama-C supports plugins which implement various kinds of analyses. We will be interested in the following ones:

- **Eva[13, 6]** computes abstract domains for variables in a program, including aliases. It can then use this information to prove (green) or disprove (red) properties expressed in ACSL. Note that proofs may be inconclusive (orange).

- **Pdg**[7] computes intra-procedural memory, data and control dependencies as dependency graphs. It is based on abstract domains computed by Eva.

Lazart: Lazart[25, 14] is a multi-fault analysis tool based on the KLEE concolic engine[15] which detects multi-fault injection attack paths. This is achieved by mutating the program to simulate faulty behavior based on some fault models and performing a dynamic symbolic execution analysis to find execution paths violating a security property. Lazart currently supports the following fault models:

- **Test Inversion:** Faults may result in the outcome of a conditional jump being inverted (*if* instructions and loop conditions).

- **Data Fault:** Faults may result in the value returned when reading a variable being altered. Users can specify which variables to fault in a strategy file. These will be made symbolic, i.e. all possible values will be considered, with the possibility of adding constraints.

These two models are very powerful as they can be used to implement other more complex models very effectively. The data fault model in particular is very generic and takes full advantage of SMT solvers to find attack paths with complex path predicates.

Tool	Results	Approximation	Precision	Scalability
Eva	abstract domains, proofs	over-approximation	low to high	robust
Pdg	intra-procedural dependency graphs	over-approximation	low to high	robust
KLEE	execution paths	under-approximation	high	path space explosion, infinite loops

Table 1: Comparing static analysis and symbolic execution

As shown on Table 1, the static analysis tools are in general less precise but scale better than symbolic execution. Additionally, since they compute over-approximations we can use them to prepare analysis with symbolic execution without risking to lose attack paths, as long as our implementation is correct. Finally, they will also terminate on any program. This is a major advantage which may help us mitigate the issue of non-terminating analyses, which is compounded by the injection of faults, compared to other approaches purely based on symbolic execution[24]. We should therefore be able to use Frama-C to counter Lazart's drawbacks.

3.2 Overview

Figure 3 gives an overview of our method which we detail afterwards.

The main argument in favor of using properties as a starting point for the analysis is that the knowledge of the code required to express them cannot be greater than that required to find attacks. This is true under the assumption that an attacker would look for the most relevant security properties to violate when attempting to attack a system, making the definition of properties a requirement for finding attacks.

Security properties can be defined manually by adding ACSL annotations to the code, however this task can be challenging if many must be introduced or if it is not obvious which ones are relevant. These difficulties can be mitigated by using automated tools to generate annotations or requiring developpers to provide a formal specification of their programs. For example, Frama-c's RTE plugin can be used to automatically add assertions for the purpose of discovering runtime errors such as buffer overflows. Another plugin, METACSL, allows to generate assertions based on high-level, user-defined properties[27].

Note that we place ourselves in the context of our implementation as a Frama-C plugin, but the concepts of our method can be used with any other tools.

3.3 Step One: Dependency Analysis

Once security properties have been defined we build a dependency graph which will allow us to find which instructions may impact them. To this end we compute a procedural dependency graph for each function using Frama-C's Pdg plugin. We then add inter-procedural dependencies by connecting the input and output nodes of these graphs to the corresponding nodes in all calls to their respective functions, which is an over-approximation of the actual inter-procedural dependencies. The program inputs should be left uninitialized to maximize coverage.

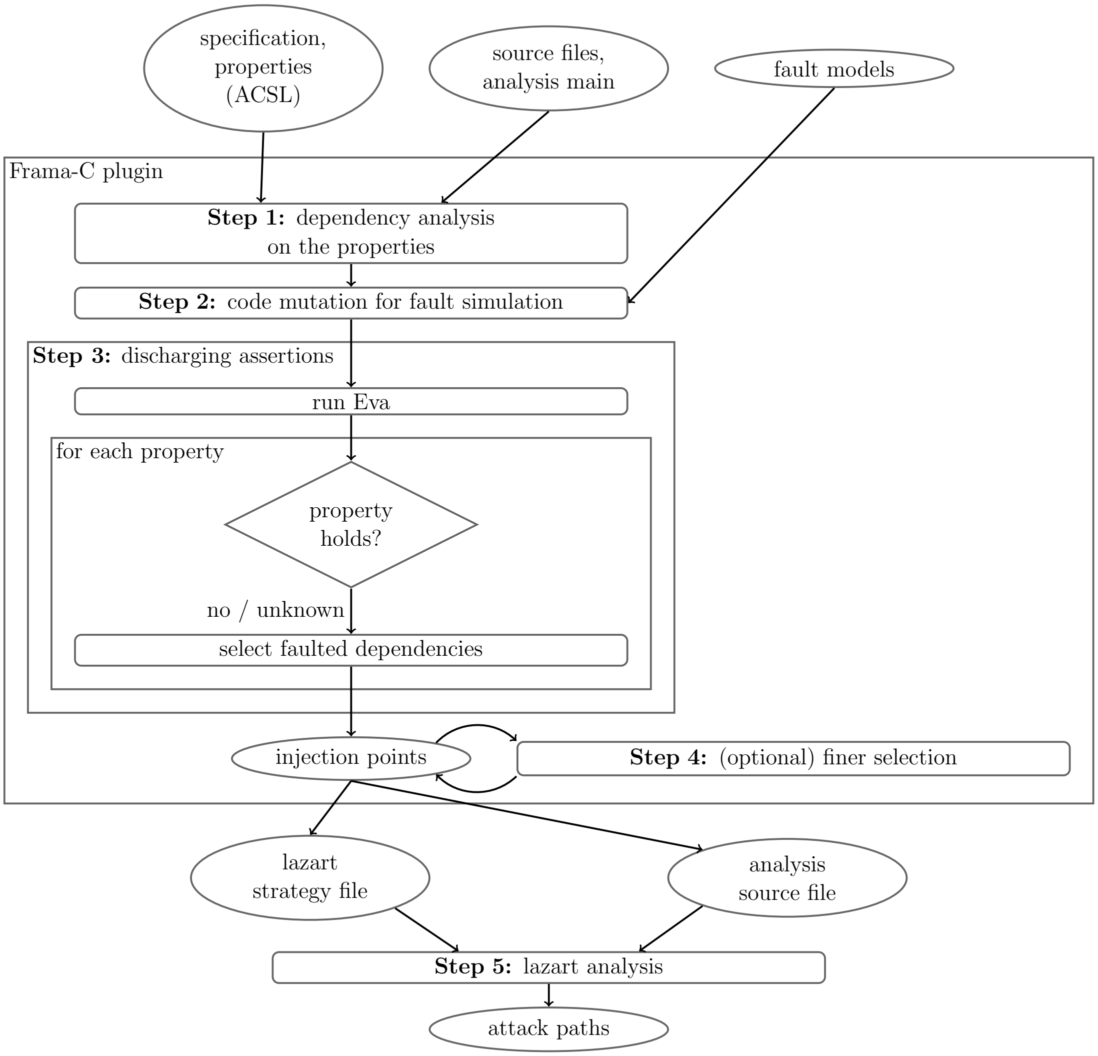

specification, properties (ACSL)
source files, analysis main
fault models
Frama-C plugin
Step 1: dependency analysis on the properties
Step 2: code mutation for fault simulation
Step 3: discharging assertions
run Eva
for each property
property holds?
no / unknown
select faulted dependencies
injection points
Step 4: (optional) finer selection
lazart strategy file
analysis source file
Step 5: lazart analysis
attack paths

Figure 3: Overview of our method

Knowing that faults may cause control flow violations (test inversion, data fault on test condition...), we must make sure that every path is explored. Control-flow instructions should therefore be treated as if their outcome is undetermined. However Pdg is too precise and does not take into account paths shown to be unfeasible by Eva when computing control dependencies. We solve this issue by making the value of test conditions arbitrary i.e. preventing Eva from being able to determine their value. This way all paths appear feasible during the dependency analysis. As shown on Figure 4, this is done by xoring an undefined extern variable (a), which Eva treats as potentially having any value (top element in the underlying lattice).

The soundness of Eva's abstract domains computation and the full exploration of execution paths regardless of feasibility under nominal conditions ensure that we do not lose attack paths. The solution that we presented is not enough to always ensure this property for any fault model however, as it does not tackle control flow graph violations such as a *then* and an *else* block

```
extern int a;
...
if(cond ^ a) //arbitrary test
```

Figure 4: Example of a test condition made arbitrary

being chained after an *if* instruction. This can happen with an *instruction skip* model which would allow to skip branching jump instructions for example. As a result our analysis in its current implementation is only suitable for fault models preserving the control flow graph of the target program.

3.4 Step Two: Simulating Faults with Eva

Being able to simulate fault with Eva is crucial as it will allow us to add faulty behavior to the target program and attempt proofs in presence of faults.

As mentioned previously, we are only considering Lazart's fault models. We combined both into a single, more powerful model which allows the value returned when computing any expression to be faulted. This is more practical in regard of simulating faults on program ASTs generated by Frama-C as we can use the same principle we used to make tests arbitrary (see Figure 4). This is also compatible with Lazart as our model is equivalent to injecting symbolic data faults on the xored variables, which we initialize to zero.

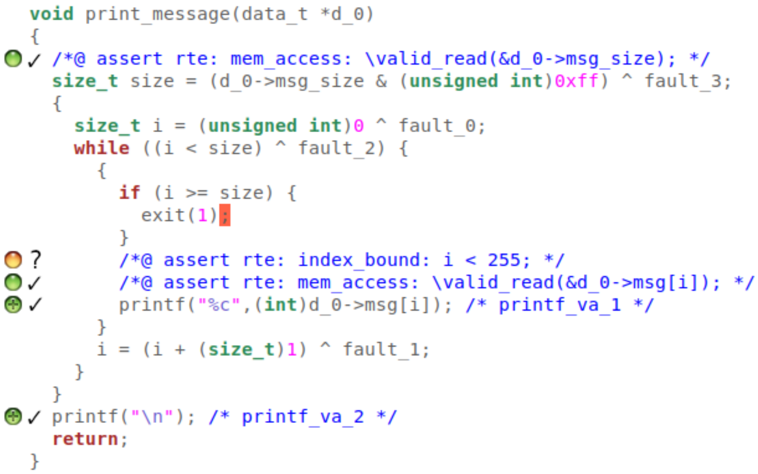
Since all instructions which contribute to a property are present in its dependency graph, we can find all the injection points relevant to that property by exploring its dependencies and selecting those which may be affected according to our fault model. We can then simulate faults on them as shown on Figure 5 (lines 9, 11, 12 and 17). In this example we used properties generated by Frama-c's RTE plugin as starting points for the analysis.

```
 1  extern unsigned int fault_3;
 2  extern int fault_2;
 3  extern size_t fault_1;
 4  extern unsigned int fault_0;
 5
 6  void print_message(data_t *d_0)
 7  {
 8    /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
 9    size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_3;
10    {
11      size_t i = (unsigned int)0 ^ fault_0;
12      while ((i < size) ^ fault_2) {
13        if (i >= size) exit(1);
14        /*@ assert rte: index_bound: i < 255; */
15        /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
16        printf("%c",(int)d_0->msg[i]); /* printf_va_1 */
17        i = (i + (size_t)1) ^ fault_1;
18      }
19    }
20    printf("\n"); /* printf_va_2 */
21    return;
22  }
```

Figure 5: Example program with simulated faults

3.5 Step Three: Discharging Assertions

In order to reduce the number of injection points to ease further analysis, we can attempt to formally prove that a property is unaffected by faults. To do this, we simulate faulty behavior on all its dependencies and then attempt to prove it using Eva. Figure 6 shows an example of proven and unproven properties on a program. If the property holds there is no need to test the associated injection points, otherwise they are selected. Note that an inconclusive proof will result in the injection points being selected to avoid losing attack paths and that Eva assumes that a property is true after its annotation (see the second *valid_read* assertion on Figure 6), which is not an issue for us.



```
    void print_message(data_t *d_0)
    {
●✓    /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
      size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_3;
      {
        size_t i = (unsigned int)0 ^ fault_0;
        while ((i < size) ^ fault_2) {
          {
            if (i >= size) {
              exit(1);
            }
●?          /*@ assert rte: index_bound: i < 255; */
●✓          /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
●✓          printf("%c",(int)d_0->msg[i]); /* printf_va_1 */
          }
          i = (i + (size_t)1) ^ fault_1;
        }
      }
●✓    printf("\n"); /* printf_va_2 */
      return;
    }
```

Figure 6: Example program with proven and unproven assertions in presence of faults

3.6 Step Four: Finer Selection

Additional selection heuristics may be used to further reduce the number of selected injection points. We used a brute force approach which consists in checking if individual injection points have an impact on the properties by setting all other fault variables to zero and running Eva, with a timeout mechanism to avoid wasting time when it struggles. This is effective as only considering one fault introduces less imprecision (see Section 4), however it is only valid in a single fault context.

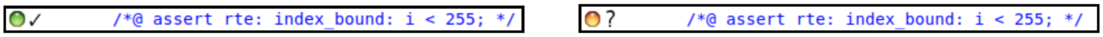
On our example program, Figure 7a shows that the *fault_0* injection point has no effect on the index bound property from line 14 on Figure 5 (the same is true for *fault_1* and *fault_2*) while Figure 7b shows that *fault_3* may have a negative impact. This is because single faults targeting the index are caught by countermeasures or restricted by the loop condition while altering the loop bound itself allows for out of bound indexes.

3.7 Step Five: Finding Attack Paths

Once we have selected injection points, we generate a strategy file for Lazart containing the corresponding fault variables. In the case of our example program, only *fault_3* is left as shown on Figure 8 since we showed it is the only injection point which may impact properties. We also output a source file containing the simulated faults.

```
extern unsigned int fault_3 = 0;
extern int fault_2 = 0;
extern size_t fault_1 = 0;
extern unsigned int fault_0;
```

```
extern unsigned int fault_3;
extern int fault_2 = 0;
extern size_t fault_1 = 0;
extern unsigned int fault_0 = 0;
```



●✓ /*@ assert rte: index_bound: i < 255; */

●? /*@ assert rte: index_bound: i < 255; */

(a) fault_0 has no effect

(b) fault_3 cannot be proven to have no effect

Figure 7: Example of an injection point with no effect and another which may impact properties

```
Parent=print_message
Fault=fault_3:symbolic
```

Figure 8: Strategy file generated for the previous example program

We finally run Lazart on these to find attack paths. We made the input message size symbolic in order to have full path coverage. Figure 9 shows the results of the analysis of our example program. As expected, we find the attack on the mask. One way this attack could be fixed would be to check that the index is smaller than 255 since the size of the buffer is fixed.

```
Fault Count              0-fault      1-fault
Injection Point
_______________________  ___________  ___________
fault_3                            0            1
```

Figure 9: Attacks found by Lazart on our example program

3.8 Our Experimental Implementation

Our Frama-C plugin fully automates all these steps. However, we also had to automate the removal of injection points causing issues during symbolic execution. As non-terminating analyses can be difficult to recognize as such with dynamic symbolic execution, the usual method to deal with this issue is to use a timeout mechanism[24]. In the case of KLEE, information about partially explored paths is available in traces and can be used to determine which injection points should be removed. We wrote a script to automatize this process and allow us to have consistent analysis times for comparisons.

In the next section we will validate our method by testing our implementation in some realistic use-cases. In particular, we will verify that we do not lose attack paths compared to selecting all injection points in the target program while exploring less executions paths and observing better performance as a result.

4 Experiments

We tested our method by analyzing several real world programs. The first one we present is the password verification part of the *sudo* unix command from Linux-PAM[4]. Our results for this program illustrate the performance benefits of our method. Then we present our analysis of the iso7816 library[2] from the ANSSI's WooKey project[3], which shows that static analysis alone can be very precise in some instances. Finally, we discuss how our method helped to discover fault attack paths bypassing countermeasures in WooKey's bootloader[8].

4.1 Sudo

We analyzed the *pam_sm_authenticate* function from Linux-PAM, which implements password verification in *sudo*, looking for paths violating the property that one cannot authenticate with a wrong password. Our goal was to show the performance benefits of our method on a fairly large program (3.6k lines of analyzed code) rather than finding attacks, which was expected given the lack of countermeasures against fault injection. This target was the best for this purpose out of all the programs we analyzed as we did not need to remove injection points causing issues with symbolic execution, meaning that the number of explored paths remained a relevant metric.

(a) On a regular computer[5].

method	injection points	analysis time		explored paths	attack paths
		Frama-C	Lazart[4]		
no selection	924	-	out of RAM	-	-
dependencies	106	10s	30s	7k	11

(b) On a cluster[6].

method	injection points	analysis time		explored paths	attack paths
		Frama-C	Lazart[4]		
no selection	924	-	1.5min	17k	10
dependencies	106	-	50s	7k	11

Table 2: Results of the analysis of sudo (data faults + test inversion)

As shown on Table 2a, our plugin allowed to reduce the number of injection points to consider almost tenfold, which in turn allowed symbolic execution to terminate correctly instead of aborting due to it running out of available memory. We did not include the results using the brute force selection heuristic as it did not make any difference on this example.

Running these analyses on a cluster[6] allowed the one with no selection step to terminate correctly. Table 2b shows that in this case we were able to reduce the number of explored paths by about 60% while finding one more attack path. This path could be lost during the control analysis due to concretization in symbolic execution.

4.2 Wookey

Both of our other targets are components of the ANSSI's WooKey project[3], a secure encrypted USB storage device requiring user authentication in order to access its content. After attacks

[4] Includes time spent removing problematic injection points.

[5] Computer specs: i5-10500 CPU (6 core, 3.10GHz), 32GB RAM

[6] Cluster specs: 2 Intel® Xeon® Gold 6138 (2GHz, 20C/40T, cache 27Mo, 10,4GT/s, 125W, Turbo, HT), 192GB RAM

were found on it by ITSEFS[11], WooKey was hardened with countermeasures, some against fault injection. However, the effectiveness of these had yet to be tested in the project's current version (0.9). We thus chose to analyze WooKey's iso7816[2] and bootloader[8] (2.5k and 3.2k lines of code respectively) in order to check that the attacks had indeed been fixed. Note that while Lazart has been used during the evaluation of WooKey's Bootloader[11], only the test inversion fault model was considered and the analysis perimeter was set manually. In contrast, we attempted to automate the discovery of complex fault attack paths with Lazart using our method.

4.2.1 Iso7816

The attack on WooKey's iso7816 library, which implements communication with a security token (a smartcard) containing cryptographic secrets, consisted in faulting a loop bound in order to cause repeated buffer overflows similarly to our motivating example. We thus chose to use memory integrity properties generated by Frama-C's RTE plugin as the starting point of our analysis. However we were unable to find any attacks on the current version of this program, which contains countermeasures in the form of index checks before any memory operation.

method	properties		injection points		analysis time		attack paths
	total	unproven	selected	used[7]	Frama-C	Lazart[4]	
no selection	-	-	656	654	-	4min	1
dependencies	384	3	151	148	2s	4min	1
dependencies + brute force	384	3	1	1	1min	2s	1

Table 3: Results of the analysis of WooKey's iso7816 (data faults, vulnerable version)[5]

For testing purposes, we reverted the originally vulnerable part of the code, in the *SC_get_ATR* function, back to its previous state. This allowed us to find the original attack as shown on Table 3. While the dependency analysis alone allowed to reduce the number of injection points considered by over 75%, the brute force selection heuristic managed to single out the injection point responsible for the violation of the three remaining properties and the attack as the relative simplicity of the properties allowed Eva to be very precise. This is illustrated by the fact that over 99% of the properties could be proven to hold even when every injection point was faulted.

4.2.2 Bootloader

The attack that was found originally on WooKey's bootloader used a fault to cause an outdated version of the firmware to be booted. As WooKey uses a dual-bank system allowing to store two firmwares (flip and flop) so that the older one can be overwritten while the other one continues to operate during updates, this attack was due to the firmware selection logic being unprotected against fault injection.

Despite countermeasures being added consisting in doubling all relevant tests, our analysis[8] shows that attack paths still exist as presented on Table 4. Using our results allowed us to identify two attacks which look feasible in practice.

[7] Injection points kept after elimination of problematic ones.

[8] the analyzed code is available as part of FISSC[9]

method	injection points		analysis time		attack paths
	selected	used[7]	Frama-C	Lazart[4]	
no selection	396	326	-	99min	12
dependencies	226	172	1.5min	76min	12
dependencies + brute force	45	34	5min	17min	12

Table 4: Results of the analysis of WooKey's bootloader (data faults + test inversion)[5]

The first attack consists in exploiting logic in the *loader_exec_req_selectbank* function, which decides which firmware to boot. Figure 10 shows a simplified version of this function. Inverting the test on line 3 when both flip and flop are bootable results in execution carrying on to the test on line 7 which only checks if the latter can be booted, assuming that one firmware at least cannot. This leads to flop being selected regardless of its version. To fix this attack, we propose to also check that flip is not bootable in that test as shown on Figure 11.

```
1  static loader_request_t loader_exec_req_selectbank(loader_state_t
       nextstate){
2      //...
3      if ((flip_shared_vars.fw.bootable == FW_BOOTABLE && flop_shared_vars.
           fw.bootable == FW_BOOTABLE) &&
4          !(flip_shared_vars.fw.bootable != FW_BOOTABLE || flop_shared_vars.
               fw.bootable != FW_BOOTABLE)){
5          //...
6      }
7      if (flop_shared_vars.fw.bootable == FW_BOOTABLE){
8          if(!(flop_shared_vars.fw.bootable == FW_BOOTABLE))
9              goto err;
10         ctx.boot_flop = sectrue;
11         //...
12     }
13     //...
14 }
```

Figure 10: Excerpt from the loader_exec_req_selectbank function in WooKey's bootloader[8]

The second attack exploits a lack of countermeasures in the *loader_exec_req_flashlock* function, which computes the pointer to the boot function of the chosen firmware. Figure 12 shows a simplified version of this function. A fault can be used to invert the test on line 4 and boot flip instead of flop. Simply doubling this test as shown on Figure 13 should be enough to solve this issue in a single fault context.

As we were only interested in the "no firmware rollback" property[9] there were no discharged assertions during this analysis. However the dependency analysis and especially the brute force selection heuristic were able to eliminate many injection points, resulting in the total duration of the analysis being reduced by as much as 75%.

4.3 Limits

Using our method can present some challenges to the user. In general, expressing properties can be difficult with limited knowledge of the code, as well as determining which ones may

[9] "the most recent firmware is booted or an error / security breach is detected"

```
static loader_request_t loader_exec_req_selectbank(loader_state_t nextstate){
    //...
    if ((flip_shared_vars.fw.bootable == FW_BOOTABLE && flop_shared_vars.fw.bootable == FW_BOOTABLE) &&
        !(flip_shared_vars.fw.bootable != FW_BOOTABLE || flop_shared_vars.fw.bootable != FW_BOOTABLE)){
        //...
    }
    if (flop_shared_vars.fw.bootable == FW_BOOTABLE && flip_shared_vars.fw.bootable != FW_BOOTABLE) {
        if(!(flop_shared_vars.fw.bootable == FW_BOOTABLE && flip_shared_vars.fw.bootable != FW_BOOTABLE))
            goto err;
        ctx.boot_flop = sectrue;
        //...
    }
    if (flip_shared_vars.fw.bootable == FW_BOOTABLE && flop_shared_vars.fw.bootable != FW_BOOTABLE) {
        //...
    }
}
```

Figure 11: Proposed fix for loader_exec_req_selectbank

```
static loader_request_t loader_exec_req_flashlock(loader_state_t nextstate){
    //...
    else if (ctx.dfu_mode == secfalse) {
        if (ctx.boot_flip == sectrue) {
            //...
            ctx.next_stage = (app_entry_t)(FW1_START);
        }
    //...
    }
}
```

Figure 12: Excerpt from the loader_exec_req_flashlock function in WooKey's bootloader[8]

be relevant targets for an attacker. Parameterizing Eva in order to be able to prove these properties is also tricky and increasing precision has a significant impact on the runtime of analyses. However the dependency analysis works well regardless of precision and is already helpful.

Multi-fault analysis also remains difficult as our brute force selection heuristic cannot be used and many injection points must still be removed for symbolic execution to terminate, by which point results tend to be irrelevant on large programs. This problem is not specific to our approach however.

Finally we designed our method with only fault models which do not allow execution paths outside of the program's original control flow graph in mind. However simulating faults before the dependency analysis would solve this issue.

```
static loader_request_t loader_exec_req_flashlock(loader_state_t nextstate){
    //...
    else if (ctx.dfu_mode == secfalse) {
        if (ctx.boot_flip == sectrue) {
            if (ctx.boot_flip != sectrue)
                goto err;
            //...
            ctx.next_stage = (app_entry_t)(FW1_START);
        }
    //...
    }
}
```

Figure 13: Proposed fix for loader_exec_req_flashlock

5 Related Works

In Christofi et al.[19] the authors attempted to prove the robustness of a CRT-RSA implementation against fault attacks using formal methods. In particular, they used Frama-C's Eva and WP plugins to prove security properties on a mutated program with simulated faults. Since their work was focused on their target specifically, they did not tackle issues that would arise when generalizing their method, namely scalability when considering realistically sized programs. Additionally, as they were only interested in formal proofs, their approach lacks the versatility required to be usable in other contexts, such as to aid with attack path detection via symbolic execution.

Our approach is similar to that used in the SANTE plugin for Frama-C[18], which uses static analysis to generate tests with alarms in order to detect runtime errors, but in the context of fault injection. SANTE is not a dedicated fault analysis tool and uses regular slicing in order to reduce the size of the tests, which makes it unfit for that particular purpose for the reasons we discussed in Section 2.2. Fault analysis may also require the generation of impractically large amounts of tests depending on the chosen fault models, which is not an issue when using symbolic execution.

SymPLFIED[24] seems to be the closest to our approach. A single symbolic variable is used to propagate the effects of fault injection using propagation rules and model checking to find attacks. Contrary to our method, SymPLFIED's approach introduces dangerous paths which are false positives (i.e fault injection that do not produces crashes or violations of security properties). Furthermore model checking is also sensitive to path space explosion and thus suffers from scalability issues.

Despite the existence of many fault analysis tools, including a few using symbolic execution, reducing the number of fault injection points to be considered in order to improve their scalability and tackle large targets with many possible execution paths is to our knowledge a novel approach.

6 Conclusion

As the need for security evaluation of not only cryptography but also critical algorithms such as authentication, bootloader and firmware update logic in embedded systems grows, so does the need for tools allowing auditors to verify their intuitions, experiment with various properties of their targets and evaluate countermeasures. These tools allow to save significant amounts of time when analyzing programs with limited insight on their inner workings. In this work we showed some solutions allowing to approach large applications, where attack paths tend to be non-trivial and can be obscured by incomplete countermeasures, using automated tools widely considered impractical in this context.

Future works could improve the links between static analysis and dynamic symbolic execution. The issue of non-terminating symbolic execution and the difficulty of multi-fault analysis should also be addressed. Finally, our approach could be applied to other tools such as fault simulators. It could also be applied at binary level by performing the static analysis part at that level or using a hybrid approach using code analysis to reduce the complexity of binary analysis.

References

[1] https://heartbleed.com. accessed july 2021.

[2] https://github.com/wookey-project/libiso7816/blob/master/smartcard_iso7816.c. patched version of iso7816, accessed july 2021.

[3] https://github.com/wookey-project. accessed july 2021.

[4] https://github.com/linux-pam/linux-pam. accessed july 2021.

[5] https://github.com/acsl-language/acsl/releases. accessed july 2021.

[6] https://frama-c.com/fc-plugins/eva.html. accessed july 2021.

[7] https://frama-c.com/download/frama-c-pdg-documentation-french.pdf. accessed july 2021, in french.

[8] https://github.com/wookey-project/bootloader/blob/master/src/main.c. accessed july 2021.

[9] https://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/proofs21.html. analyzed code, accessed september 2021.

[10] Application of Attack Potential to Smartcards and Similar Devices. Technical Report Version 3.0, Joint Interpretation Library, April 2019.

[11] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, and Trusted Labs. Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. In *SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications*, 2020.

[12] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[13] S. Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *18th International Conference on Verification Model Checking and Abstract Interpretation (VMCAI 2017)*, volume 10145 LNCS of *Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation*, pages 112–130, Paris, France, January 2017.

[14] Etienne Boespflug, Cristian Ene, Laurent Mounier, and Marie-Laure Potet. Countermeasures Optimization in Multiple Fault-Injection Context. In *"Fault Diagnosis and Tolerance in Cryptography" FDTC 2020*, Milan (Virtual Workshop), Italy, September 2020.

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[16] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56:82–90, 02 2013.

[17] The CCRA Management Committee. Common Criteria for Information Technology Security Evaluation, September 2012.

[18] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging. In Martin Gogolla and Burkhart Wolff, editors, *5th International Conference on Tests & Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 78–83, Zurich, Switzerland, June 2011. Springer Verlag. The original publication is available at www.springerlink.com.

[19] Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptogr. Eng.*, 3(3):157–167, 2013.

[20] Ang Cui and Rick Housley. BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[21] Patrice Godefroid, Michael Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10:20, 03 2012.

[22] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[23] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[24] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481, 2008.

[25] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation*, Cleveland, United States, March 2014.

[26] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne, and Maxime Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In *Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, volume 8930 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[27] Virgile Robles, Nikolai Kosmatov, Virgile Prévosto, Louis Rilling, and Pascale Le Gall. Methodology for Specification and Verification of High-Level Requirements with MetAcsl. In *FormaliSE 2021 - 9th International Conference on Formal Methods in Software Engineering*, Online conference, France, May 2021. IEEE TCSE and SIGSOFT.

[28] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined fault and side-channel attack on protected implementations of aes. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, pages 65–83, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[29] Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c: a software analysis perspective. *Formal Aspects of Computing*, 27:573–609, 10 2012.

[30] F. Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3, 1995.

[31] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 41–48. IEEE Computer Society, 2017.