Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities

Guilhem Lacombe, David Feliot, Etienne Boespflug, Marie-Laure Potet

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

Guilhem Lacombe[1,2*], David Feliot[2], Etienne Boespflug[1,3] and Marie-Laure Potet[1,3]

[1]Université Grenoble Alpes, Grenoble, France.
[2]LETI CESTI, CEA, Grenoble, France.
[3]Verimag, Grenoble, France.

*Corresponding author(s). E-mail(s): guilhem.lacombe@cea.fr;
Contributing authors: david.feliot@cea.fr; etienne.boespflug@univ-grenoble-alpes.fr;
marie-laure.potet@univ-grenoble-alpes.fr;

Abstract

Certification through auditing allows to ensure that critical embedded systems are secure. This entails reviewing their critical 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. 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 [19]. 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 [11]. Included in their arsenal are perturbation attacks, also known

This work is partially supported by ANR-15-IDEX-02. All data generated or analysed during this study are included in this published article.

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 [13, 28]. This can lead to secret information such as bits of cryptographic keys leaking through side channels [37] or even to loading an outdated firmware [23, 40].

The tasks auditors must fulfill include reviewing the critical components of the target program such as cryptographic code, bootloaders or authentication procedures by exploring execution paths looking for dangerous behaviors, for example allowing to bypass said cryptographic code. 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 [18, 26], 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, discuss some additional heuristics in Section 4 and provide experimental validation in Section 5. Finally, we discuss related works in Section 6.

2 Context and Contributions

2.1 Security Evaluation and Fault Analysis

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 analyze them first in order to familiarize themselves with the target. Potential attacks can be detected based on fault models (i.e. known high-level effects of hardware faults), giving insight on what parts of the code are likely to be vulnerable to fault injection as it has been shown that a combination of a powerful fault model and multi-fault analysis effectively covers low-level attacks [35]. This information can then be used by developers to decide where additional countermeasures are needed as well as to guide further evaluation at instruction and hardware levels. Thus source-level fault analysis is an important first step in evaluating the resistance to fault injection attacks of a program despite being often overlooked.

Fault analysis is often associated with the evaluation of cryptographic implementations. While fault attacks are a major threat in that context, another equally dangerous application of fault injection is to disrupt program logic outside of cryptographic components, sometimes bypassing them entirely. For example, faults have been used to alter the control-flow of bootloaders and perform exploits on real hardware [16].

The main distinction between these two applications of fault injection is that the types of faults considered differ. If the fault models chosen in a crytographic context tend to focus on altering data, for example by allowing to set values to zero, those used for evaluating non-cryptographic code focus more on altering control-flow as it is often more logic- and decision-centric rather than computation-heavy. As a result the skillsets required to evaluate these two types of code are

different and thus they are handled by separate teams.

In this work we focus primarily on the evaluation of non-cryptographic programs with complex control-flow resulting in a large set of possible execution paths, which grows exponentially when faults are taken into account. However we also discuss its application in a cryptographic setting at the end of section 5.

2.2 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.

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 inadvisable, 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 would therefore allow to not only more reliably detect fault injection vulnerabilities, but also to evaluate the effectiveness of countermeasures.

2.3 Usage of Tools in Security Evaluation

As part of their job, security evaluators must discover vulnerabilities within a limited time frame while taking into account the current state-of-the-art. Automated tools are therefore of interest to them as they allow to speedup analysis while offering specific guarantees in terms of coverage. Another point of interest is that they allow to easily replay analyses, rendering cross-checking and updating results much more efficient.

Using such tools presents its own set of challenges however, as parameterizing them, creating attack scenarios and interpreting results all require extensive knowledge and experience to be done correctly. One particular difficulty of interest to us is the definition of an analysis perimeter, which is an area of the target program containing relevant elements toward the chosen attack scenario that the analysis will be restricted to, with the intention of improving scalability. This is especially important when analyzing large programs which tools may struggle to handle if considered in their entirety. For example, evaluators may want to only analyze a few functions while stubbing the rest, with restrictions on inputs.

Ideally, a sound approach to extracting an analysis perimeter should be favored in order to ensure that no attack will be missed. However this is often impractical when tools are suffering from scalability issues, which may cause the analysis to not terminate within a reasonable time frame or precision to be insufficient. A solution is to progressively restrict the analysis perimeter based on heuristics, but since it can be difficult to judge the progress of analyses, determining when to interrupt them and apply further restrictions is a blind process. For this reason, a perimeter widening approach consisting in progressively analyzing more code with less restrictions is often preferred as previous terminating analyses give a frame of reference for forming expectations on runtime.

[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 [12], section 9.

[3] See the SC_get_ATR function.

```
1  typedef struct data{
2    size_t msg_size;                    //input > 255
3    char msg[256];                      //input
4    uint32_t key[8];                    //secret
5  } data_t;
6
7  void print_message(data_t *d){
8    unsigned int size = d->msg_size & 0xff; //countermeasure, fault 0x000000ff to 0xffffffff
9    for(unsigned int i = 0; i <= size; i++){
10     if(i > size)                      //bypassed countermeasure
11       return;
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 }
```

Fig. 1: Example of a function vulnerable to fault injection

In this context, using scalable but less precise static analysis techniques in order to guide the usage of more precise but less scalable dynamic ones is becoming common practice [24].

2.4 Program Analysis Techniques

Many program analysis methods and tools can be used for security evaluation. The following is a succinct presentation of commonly used ones which are relevant to this work.

Abstract Interpretation

Abstract Interpretation is a static analysis theory for the sound approximation of program semantics applied to ordered domains such as lattices. It is used to gain some insight on the properties of the program without fully executing it. Applications include computing value ranges for variables, data-flows and dependencies as well as checking assertions. Implementations tend to scale well to realistically sized programs, although with stubbed functions and some loss of precision.

Dynamic Symbolic Execution (DSE)

DSE is a dynamic analysis technique consisting in exploring execution paths in a program while constructing constraint formulas for variables with unspecified values, which are referred to as symbolic variables. These constraints can then be solved using SMT solvers to check the feasibility of execution paths and to obtain inputs triggering them. DSE implementations tend to have scalability issues when constraint formulas become too complex or due to path space explosion. They can also get stuck in infinite loops and never terminate.

Concolic execution is a variant which allows to concretize symbolic variables (i.e. give them a concrete value) to mitigate these issues. However this is not always enough and may induce a loss of correctness and completeness [25], i.e. some explored paths may not actually be feasible and some feasible paths may not be explored.

2.5 Difficulties of Fault Analysis

Fault analysis is particularly tricky as most widely used analysis techniques and tools are not designed with faults in mind. The main issue is that faulty behavior must be accounted for at various program points, which we refer to as *fault injection points*. This introduces a large number of new feasible execution paths, which increases exponentially with the number of faults considered. Thus the choice of the analysis perimeter becomes crucial as it directly impacts this issue.

Extracting an analysis perimeter is usually done by expressing assertions related to security properties and slicing [27, 39] based on dependencies. This allows to produce a minimal program corresponding only to relevant execution paths with regard to the 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, especially in a multi-fault context.

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

```
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 }
```

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

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. Slicing may also result in countermeasures being removed since they often appear as redundant or dead code under nominal conditions.

Another issue with slicing is that code may be modified to the point that making the connection to the original is difficult. This can make the interpretation of results difficult as well as resuce trust in the accuracy of the analyses.

The consequence of these difficulties is that reducing the size of the analyzed program in any meaningful way 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.

2.6 Contributions

- We propose an approach using formal methods to find the injection points that a security property depends on and eliminate those which can be proven to have no impact. Faulty behavior is simulated using a generic fault model adapted to source-level analysis. Relevant injection points are then selected for further analysis with a dedicated fault analysis tool.
- We implement 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 implemented as a Frama-C plugin. Our fault analysis tool of choice is Lazart, which is itself based on dynamic symbolic execution by KLEE.
- We present and test various heuristics allowing to deal with the scalability issues that arise when using dynamic symbolic execution for fault analysis.
- 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 verifyPIN program from the FISSC fault injection test suite [4] and the *sudo* unix command [5].

This paper presents multiple extensions of our original work.

- We revised our method to streamline it, with a greater emphasis on our generic fault model. Our implementation was also updated to use the latest version of Lazart.
- New selection heuristics designed to help with the scalability of analyses were added and tested.

- We tested our method on new targets from FISSC and validated our proposed fixes to WooKey's bootloader against single-fault attacks. We also showed that these new countermeasures do not hold against double-fault attacks.

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 when eliminating false positives using dynamic symbolic execution, as illustrated in section 5.

3 Our Method

Our goal with this work is to design a method allowing to improve the scalability of fault analysis tools by reducing the number of injection points considered without risking to remove important parts of the target program nor losing attack paths. We chose to use static source code analysis as a front-end in order to help with the evaluation of code-level countermeasures. Additionally this helps to approach unknown code as the relevant parts to the targeted security properties can be easily pointed out. Dynamic symbolic execution analysis then provides more precise information which is easier to interpret in this context.

3.1 Tools

We based our method and its implementation on proven and widely used tools, which need to be discussed as they impacted design decisions. It is however important to note that the general concepts of our method should be applicable to any other tools.

Frama-C

Frama-C [38] is a static analysis platform for the C language. It parses .c files into a formal AST structure (based on CIL) which supports annotations detailing specifications and properties using the ACSL specification language [6]. Frama-C then manages plugins which implement various kinds of analyses. We will be interested in the following ones:

- **Eva** [7, 14] performs abstract interpretation and computes abstract domains for variables in a program, including aliases. It can then use this information to correctly prove or disprove assertions expressed in ACSL, with inconclusive attempts being labeled as such. Eva is thus useful not just for bug-finding but also for proving properties.
- **Pdg** [8] computes intra-procedural memory, data and control dependencies as dependency graphs. It is based on data-flow analysis using Eva, which allows for greater precision as values can be taken into account.
- **From** computes inter-procedural dependencies using data-flow analysis through Eva.

Lazart

Lazart [15, 33] is a multi-fault analysis tool based on the KLEE concolic engine [17] which detects multi-fault injection attack paths. This is achieved by mutating the program using a Clang compilation pass in order to simulate faulty behavior based on some fault models. A dynamic symbolic execution analysis is then performed at LLVM IR level to find execution paths violating a security property. Correctness and completeness are inherited from KLEE, thus false positives may appear or attack paths may be lost if variables are concretized. Lazart is a state-of-the-art tool with concrete use-cases in the context of fault analysis and security evaluation by ITSEFs [12].

As previously discussed, dynamic symbolic execution suffers from scalability issues mainly due to path space explosion. For this reason large programs with many potential fault injection points are difficult to analyze with Lazart, especially in a multi-fault context. However since the static analyses implemented by Frama-c's plugins are based on abstract interpretation, they are less prone to these issues and can thus be used to mitigate those from Lazart.

3.2 A Generic Fault Model

Inferring the effect of hardware level faults on programs is very difficult as it would require a full understanding of the target platform from micro-architectural details to its firmware's implementation. For this reason, most fault analysis techniques rely on fault models to determine faulty behaviors that may occur during execution.

Commonly considered fault models include the following:

- **Data Load:** The value obtained when reading a variable can be altered (e.g. set to zero).
- **Test Inversion:** The outcome of a conditional jump can be inverted.
- **Control-flow Violations:** Instructions may be skipped and jumps may lead to unexpected locations.

These do not correspond to distinct types of faults but are rather higher level interpretations of faulty behavior which may overlap depending on scenarios. For example, faulting the value of a test condition (data load model) is equivalent to inverting the test itself (test inversion model). The data load and test inversion models are particularly relevant to us since they are available in Lazart.

Given our goal of evaluating at source level which potential fault injection points in a program are worth considering, we should choose a fault model generic enough to be relevant toward those previously described. We should also take into account the limitations of source analysis, namely the lack of information on the runtime memory and the compiled binary layouts. Our fault model should therefore be restricted to faulty behaviors which can be statically processed without this information. For example, it should not allow to skip control instructions such as conditional jumps as what code would be executed next cannot be inferred at source level. This is acceptable as such behavior would be extremely difficult to explore with current fault analysis tools. Additionally some scenarios can be simulated in more source-friendly ways e.g. skipping function calls can be done by encapsulating them in tests.

We thus propose the following fault model:

- **Expression Fault:** The evaluation of an expression can be altered. "Expression" refers here to *exp* objects in the CIL AST, which correspond to C expressions as defined in the C standards minus assignments[4].

Additionally, we restrict our fault model to non-pointer expressions[5]. This, along with the fact that control instructions cannot be skipped, ensures that the control-flow graph of the program is never violated.

Figure 3 shows how this fault model affects our motivating example. Our fault model encompasses the effects of data load faults (lines 8, 9 and 12) and allows for test inversion since test conditions can be directly faulted (lines 9 and 10). The effect of skipping instructions and faulting pointers on non-pointer data is also covered. For example, skipping the loop counter incrementation on line 9 is equivalent to faulting the value of $i + 1$. However these behaviors may require multiple faults to be covered in general e.g. to cover the effect of skipping a call to a function with implicit outputs (i.e. assignments to global variables, input pointer memory...) expressions containing them should all be faulted.

This model also encompasses faulty behavior which the others do not cover, such as errors occurring during computation of operations, e.g. obtaining an odd result from a multiplication by two.

3.3 Overview of our Method

Figure 4 gives an overview of our method, which takes a source file with security properties expressed in ACSL as input. The static analysis part is performed automatically by our Frama-C plugin and its outputs are immediately ready to be analyzed with lazart.

While properties can be difficult to extract from unknown code, the knowledge required to express them cannot be greater than that required to find attacks. This is assuming that an attacker would look for the most relevant security properties to violate rather than proceed blindly 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 assertions to the potentially vulnerable 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 assertions or requiring developers 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

[4] In this case, only the right-side of the assignment is translated to *exp*.

[5] Note that array indexes can still be faulted.

```
1  typedef struct data{
2    size_t msg_size;
3    char msg[256];
4    uint32_t key[8];
5  } data_t;
6
7  void print_message(data_t *d){
8    unsigned int size = fault(d->msg_size & 0xff);
9    for(unsigned int i = fault(0); fault(i <= size); i = fault(i + 1)){
10     if(fault(i > size))
11       return;
12     printf("%c", fault(d->msg[i]));
13   }
14   printf("\n");
15 }
```

Fig. 3: Expression fault model applied to our motivating example

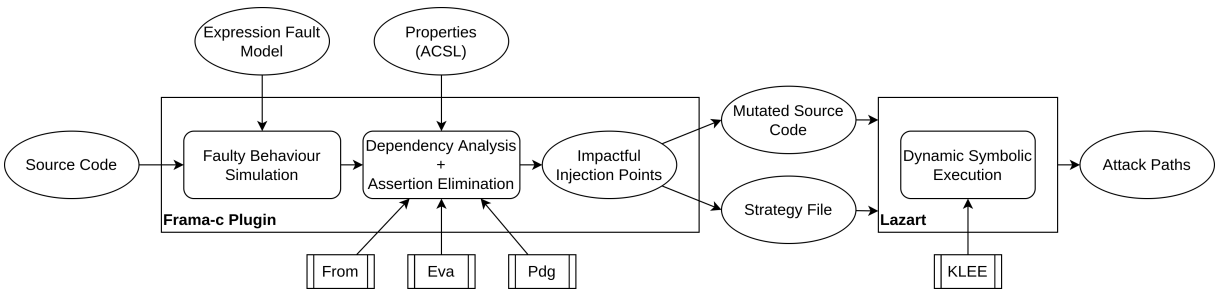

Fig. 4: Overview of our method

runtime errors such as buffer overflows. Another plugin, METACSL, allows to generate assertions based on high-level, user-defined properties [36].

3.4 Introducing Faulty Behavior

As faulty behavior must be taken into account when analyzing programs, the first step we have to take is to simulate it everywhere indiscriminately. Then we will be able to conduct further analysis and select the most important injection points. As Eva is the basis for all of the Frama-c analyses we will rely on, we thus need to find a way to force value ranges computed for expressions to be imprecise in order to enforce our fault model.

As shown on Figure 5, this is done by xoring undefined extern variables (e.g. *fault_2*), which we refer to as *fault variables*, to expressions (e.g. to the test condition on line 14). Eva then treats the fault variable, and thus the entire expression, as potentially having any value. In the case of nested expressions only the top-level is faulted as this will include the effects of faulting sub-expressions.

3.5 Dependency Analysis

Once security properties have been defined and faults simulated we build a dependency graph for the program in order to find the instructions impacting each assertion. To this end we compute a procedural dependency graph for each function using Frama-C's Pdg plugin [8]. We then add inter-procedural dependencies by connecting the input and output nodes of these graphs to the corresponding nodes in calls to their respective functions obtained using Frama-C's From plugin. As Pdg and From are based on data-flow analysis from Eva, the previously simulated faults are taken into account.

All relevant injection points to an assertion are found by exploring the dependency graph starting from its node, selecting those encountered on dependencies. This way, *fault_5* on Figure 5 (line 18) is determined to not have any impact on the assertions on lines 10, 16 and 17 (which were generated using Frama-c's RTE plugin). We rely on the soundness of Eva's abstract domains computation to ensure that we do not lose dependencies

```
1  extern int fault_5;
2  extern unsigned int fault_4;
3  extern int fault_3;
4  extern int fault_2;
5  extern size_t fault_1;
6  extern unsigned int fault_0;
7
8  void print_message(data_t *d_0)
9  {
10   /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
11   size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_4;
12   {
13     size_t i = (unsigned int)0 ^ fault_0;
14     while ((i <= size) ^ fault_2) {
15       if ((i > size) ^ fault_3) goto return_label;
16       /*@ assert rte: index_bound: i < 256; */
17       /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
18       printf("%c",(int)d_0->msg[i] ^ fault_5);
19       i = (i + (size_t)1) ^ fault_1;
20     }
21   }
22   printf("\n");
23   return_label: return;
24 }
```

Fig. 5: Motivating example with simulated faults

```
  void print_message(data_t *d_0)
  {
ok /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
   size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_4;
   {
     size_t i = (unsigned int)0 ^ fault_0;
     while ((i <= size) ^ fault_2) {
       {
       if ((i > size) ^ fault_3) {
         {
           goto return_label;
         }
       }
?      /*@ assert rte: index_bound: i < 256; */
ok     /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
ok     printf("%c",(int)d_0->msg[i]); /* printf_va_1 */
       }
       i = (i + (size_t)1) ^ fault_1;
     }
   }
ok printf("\n"); /* printf_va_2 */
   return_label: return;
  }
```

Fig. 6: Motivating example with proven and unproven assertions in presence of faults

and preserve all attack paths conforming to our fault model.

At this point, we can try to eliminate more injection points by proving that assertions are verified in presence of faults using Eva. Figure 6 shows which assertions can be proven or not on our motivating example. Injection points which are only dependencies of proven assertions can be discarded. Assuming that Eva's proofs are indeed correct, this ensures that no attack path is lost. Note that Eva assumes that an assertion is true after its annotation (see the second *valid_read* assertion on Figure 6).

3.6 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 motivating example we are left with five of them as shown on Figure 7. We also output a source file containing the simulated faults.

```
version: 4.0.0
tasks:
  add_trace:
    - __mut__
  rename_bb:
    - __mut__
  countermeasures: []
fault-space:
  functions:
    __all__:
      models:
      - type: data
        vars:
          fault_4: __sym__
          fault_3: __sym__
          fault_2: __sym__
          fault_1: __sym__
          fault_0: __sym__
```

Fig. 7: Strategy file generated for the previous example program

We finally run Lazart to find attack paths, targeting the fault variables with the data-load fault model. Figure 8 shows the results of the analysis, which indicate that only the fault on the masking

of the loop bound (line 11 on Figure 5) is dangerous in a single fault context. One way this attack could be fixed would be to check that the index is smaller than 256 since the maximum size of the buffer is fixed.

Fault Count Injection Point	0−fault	1−fault
fault_4	0	1

Fig. 8: Attacks found by Lazart on our example program

While our method mitigates the scalability issues of symbolic execution, it is sometimes not enough as shown in our experimentation. In such instances results can be obtained by weakening our fault model or eliminating more injection points based on heuristics, which may result in a loss of completeness. In the case of our motivating example we had to use a fixed value for the size of the message as making it symbolic caused the analysis to be unreasonably slow for such a small program. In the following section we will discuss a few injection point selection heuristics which can be used to remediate this issue.

4 Additional Selection Heuristics

While our method helps to mitigate the scalability issues of symbolic execution, it does not fully eliminate them. Thus the use of additional selection heuristics is often necessary to further reduce the number of selected injection points and obtain results within a reasonable time frame.

4.1 Brute-Force Selection

The brute-force selection approach consists in checking if individual injection points have an impact on the assertions 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 improves precision, however this is only valid in a single fault context. For multi-fault analysis all combinations of injections points would have to be considered, which is not practical as the number of tests that would need to be conducted increases exponentially with the maximum number of faults allowed.

On our motivating example, Figure 9a shows that the *fault_0* injection point has no effect on the index bound property from line 16 on Figure 5 (the same is true for *fault_1*, *fault_2* and *fault_3*) while Figure 9b shows that *fault_4* 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. *fault_4* is therefore the only injection point left to analyze with Lazart and the complexity of the analysis is drastically reduced, allowing us to test with a symbolic message size. Using the brute-force selection heuristic thus allowed us to ensure completeness in this case. Note that our plugin removes inactive faults from tests conditions as we noticed they impacted the precision of the analysis.

4.2 Limiting Fault Injection Point Occurrences

While fault injection points are assimilated to program locations for the purposes of static analysis, during execution these may correspond to multiple instances where faulty behavior may occur. For example, an single injection point within a loop corresponds to as many potential faults as the maximum number of iterations during execution.

Fault injection points with high occurrence rates are more likely to induce scalability issues as each occurrence generates new paths to explore. We could thus use an analysis strategy consisting in testing low occurrence injection points first hoping that the analysis terminates in a reasonable amount of time on the first try. The analysis perimeter can then be expanded by adding higher occurrence injection points until further analysis becomes impractical.

We can obtain occurrence counts during a normal execution by computing the maximum value of counters placed before each injection point using Eva while all faults are inactive. Figure 10 shows our motivating example with injection point occurrence counters and their maximum value. In this case, the one we showed was dangerous in the previous section, *fault_4*, only occurs once. An analysis only over low occurrence injection points

```
1  unsigned int fault_4 = 0;
2  int fault_3 = 0;
3  int fault_2 = 0;
4  size_t fault_1 = 0;
5  extern unsigned int fault_0;
```

```
  void print_message(data_t *d_0)
  {
ok /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
   size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_4;
   {
     size_t i = (unsigned int)0 ^ fault_0;
     while (i <= size) {
       {
         if (i > size) {
           {
             goto return_label;
           }
         }
ok       /*@ assert rte: index_bound: i < 256; */
ok       /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
ok       printf("%c",(int)d_0->msg[i]); /* printf_va_1 */
       }
       i = (i + (size_t)1) ^ fault_1;
     }
   }
ok printf("\n"); /* printf_va_2 */
   return_label: return;
  }
```

(a) *fault_0* has no effect

```
1  extern unsigned int fault_4;
2  int fault_3 = 0;
3  int fault_2 = 0;
4  size_t fault_1 = 0;
5  unsigned int fault_0 = 0;
```

```
  void print_message(data_t *d_0)
  {
ok /*@ assert rte: mem_access: \valid_read(&d_0->msg_size); */
   size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_4;
   {
     size_t i = (unsigned int)0 ^ fault_0;
     while (i <= size) {
       {
         if (i > size) {
           {
             goto return_label;
           }
         }
 ?       /*@ assert rte: index_bound: i < 256; */
ok       /*@ assert rte: mem_access: \valid_read(&d_0->msg[i]); */
ok       printf("%c",(int)d_0->msg[i]); /* printf_va_1 */
       }
       i = (i + (size_t)1) ^ fault_1;
     }
   }
ok printf("\n"); /* printf_va_2 */
   return_label: return;
  }
```

(b) *fault_4* cannot be proven to have no effect

Fig. 9: Example of an injection point with no effect and another which may impact properties

would therefore find all single-fault attack paths in this program.

In the case of multi-fault analysis, injection point occurrences should ideally be computed with simulated faults. However this yields results too imprecise to be of use in our experience. The single-fault method can still be used as injection points which are troublesome in single-fault are even more so in multi-fault, however we are missing occurrence data for injection points located in dead code, which may become reachable due to an earlier fault.

We can try to infer "projected" occurrences for these in order to get a selection criteria: when a conditional jump is encountered and one target is dead code, the occurrence count of the jump is attributed to those in the dead code which can be inferred to occur once in an execution of the dead code.

Figure 11 shows a conditional jump with a dead target, occurring n times. In the dead code *fault_1* and *fault_3* have n projected occurrences as they occur once in an isolated execution of that code, while *fault_2* has no projected occurrences as such execution can loop and visit it multiple times. We can thus infer that *fault_1* and *fault_3* are less likely to cause scalability issues than *fault_2*.

4.3 Strategy Shrinking and Growing

Aside from static analysis, various analysis strategies can be used to detect which injection points cause performance issues during analysis with Lazart.

Strategy shrinking consists in interrupting the analysis after a set amount of time and analyzing KLEE's traces corresponding to unfinished path explorations[6] in order to find which injection points are triggered the most in them. These are then assumed to cause performance issues and removed from the strategy file. This method can be effective if only a few injection points are problematic, however it is difficult to use properly as one need to guess how long the analysis should be running before interrupting. It is also not compatible with multi-fault analysis as in that case it is not clear which of the multiple triggered injection points within a trace is causing issues.

Strategy growing is a similar method consisting in adding injection points to the analysis one-by-one, keeping them if the analysis then terminates within the allowed time frame. This approach is more reliable as previous terminating analyses give a point of reference to set timeouts i.e. an analysis with one more injection point can be expected to terminate within the same time as the previous one plus some margin. For this reason

[6] Traces with a .early extension.

```
1  unsigned int fault_4 = 0;
2  int fault_3 = 0;
3  int fault_2 = 0;
4  size_t fault_1 = 0;
5  unsigned int fault_0 = 0;
6
7  unsigned int fault_0_counter = 0;
8  unsigned int fault_1_counter = 0;
9  unsigned int fault_3_counter = 0;
10 unsigned int fault_2_counter = 0;
11 unsigned int fault_4_counter = 0;
12
13 void print_message(data_t *d_0)
14 {
15   fault_4_counter ++;
16   size_t size = (d_0->msg_size & (unsigned int)0xff) ^ fault_4;
17   {
18     fault_0_counter ++;
19     size_t i = (unsigned int)0 ^ fault_0;
20     while (1) {
21       fault_2_counter ++;
22       if (! ((i <= size) ^ fault_2))
23         break;
24       fault_3_counter ++;
25       if ((i > size) ^ fault_3)
26         goto return_label;
27       printf("%c",(int)d_0->msg[i]);
28       fault_1_counter ++;
29       i = (i + (size_t)1) ^ fault_1;
30     }
31   }
32   printf("\n");
33   return_label: return;
34 }
```

(a) Motivating example with fault occurrence counters

Injection Point	fault_0	fault_1	fault_2	fault_3	fault_4
Occurrences	1	256	257	256	1

(b) Occurrences of each injection point

Fig. 10: Occurrences of injection points in our motivating example

```
1  if(true ^ fault_0){          //n occurrences
2      ...
3  }
4  else{
5      int x = 0 ^ fault_1;     //n projected occurrences
6      while(... ^ fault_2){    //no projection
7          ...
8      }
9      x = 1 ^ fault_3;         //n projected occurrences
10 }
```

Fig. 11: Example of projected occurrences in dead code

it is better suited for multi-fault analysis, although the order in which injection points are introduced may impact which ones are kept. For example, if one injection point is problematic when coupled with n others and is added first, then the n others will be rejected. However if these n points were added first then only the former may be rejected. Note that in the case of single fault analysis the injection points can be tested individually to save time.

Table 1: Characteristics of the Selection Heuristics

Heuristic	Fault Context	Completeness
Brute-force Selection	single	✓
Occurrence Limit	single	-
Strategy Shrinking	single	-
Strategy Growing	multi	-

4.4 Conclusion

To our knowledge, there is no miracle solution to the scalability issues of symbolic execution, especially when treating it as a black box. As heuristics are inherently unreliable, maintaining a diverse toolbox of them is key to be efficient. Table 1 shows if those we discussed are valid in single- or multi-fault contexts and whether they incur a loss of completeness. In general the static analysis based ones are faster and should therefore be tried first before additionally resorting to strategy shrinking or growing.

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.

5 Experiments

We tested our method by analyzing a commonly considered example from the FISSC fault injection test suite [4] as well as several real world programs. In this section we first showcase our method on the verifyPIN example from FISSC in order to illustrate its benefits. We then present our analysis of the password verification part of the *sudo* unix command from Linux-PAM [5]. Next we analyze the iso7816 library [2] from the ANSSI's WooKey project [3] and show that static analysis alone can be very precise in some instances. Finally, we discuss how our method helped to discover single-fault attack paths bypassing countermeasures in WooKey's bootloader [9], propose some fixes and test their effectiveness in single- and double-fault contexts.

We used the injection point selection heuristics discussed in the previous section when necessary.

Table 2: Results of the analysis of verifyPIN (single faults)

(a) without countermeasures

Static Analysis		IP	Lazart Analysis		
Deps	Time		AP	EP	Time
-	-	20	5	65	14s
✓	1s	10	5	52	11s

(b) with Lalande's countermeasure

Static Analysis		IP	Lazart Analysis		
Deps	Time		AP	EP	Time
-	-	142	6	3 253	1h 43min
✓	1s	15	6	1 051	30min

Deps: Dependencies
IP: Injection Points
AP: Attack Paths
EP: Explored Paths

We preferred strategy growing over shrinking, despite it being slower in some cases, as it is less reliant on guesswork and is thus more appropriate in scenarios where evaluators have limited knowledge of their target.

The main criteria we use to evaluate our method are the number of injection points left to analyze with Lazart and the time required to obtain results. Symbolic execution metrics such as the number of explored paths are also considered.

5.1 VerifyPIN

FISSC [4] is a collection of programs designed to test fault analysis techniques. It is mainly composed of variants of verifyPIN, a small PIN code verification program, with different countermeasures applied to it. In a single fault context, we analyzed the basic variant without countermeasure and one with Lalande's countermeasure [29], which consists in propagating and checking an instruction counter to reliably detect skips of at least two instructions. While this countermeasure is not useful against our fault model, which does not allow for violations of the control-flow graph of the program, it adds a lot of complexity to the analysis.

Table 2 shows the results of our analysis. As expected, some attacks paths were found where

the attacker was able to authenticate with an incorrect PIN code.

As shown in the first part, our method yielded a small improvement in terms of runtime and explored paths during symbolic execution in the case of the basic version of verifyPIN. The most significant difference lies with the number of injection points considered, which is halved. This is helpful to evaluators trying to interpret the results of the analysis as only relevant parts of the target have injection points as opposed to everything indiscriminately.

The second part shows that the dependency analysis allowed to eliminate most injection points introduced by Lalande's countermeasure, resulting in the number of injection points to consider being reduced by around 90%. This then induces a greater than 66% reduction in the number of explored paths and the duration of analysis with Lazart, for a negligible cost.

Finally, we find the same number of attack paths in both instances between the two approaches. Our method thus did not induce a loss of attack paths relatively to our fault model in this case.

5.2 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. In this case we had to use strategy growing in order to remove a few injection points causing path explosion issues.

Table 3 shows that the dependency analysis allowed to reduce the number of injection points to consider by almost 90%, which then directly impacts the duration of the strategy growing step as each one is tested separately. Our approach is thus very beneficial to use this heuristic effectively as it allowed to reduce the analysis time by roughly 85% in this case, while no attack paths were lost in the end compared to not using it.

We also tested injection points based on their occurrences as shown in the second part of Table 3. We found that limiting occurrences to one yields results fast, although not all attack paths are discovered. However increasing the limit results in problematic injection points being added and thus strategy growing would have to be employed to test the remaining ones.

5.3 Wookey

Both of our last two 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 were found on it by ITSEFS [12], 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 [9] (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 [12], 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.

5.3.1 Iso7816

The attack on WooKey's iso7816 library, which implements communication with a security token (a smartcard) containing cryptographic secrets, consisted in using a single fault to modify 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. We also chose to disable the test inversion part of our fault model as such faults usually only lead to off-by-one overflows.

As we found no attack path with a single fault on the current version of iso7816, 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 4. 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

Table 3: Results of the analysis of sudo (single faults)

(a) without limits on injection point occurrences

Deps	BF	Static Analysis		Strategy Growing		Final Lazart Analysis		
		IP	Time	IP	Time	AP	EP	Time
-	-	-	-	919 → 913	23min	17	737	11s
✓	-	104	1s	104 → 102	3min	17	670	11s

(b) with limits on injection point occurrences

Deps	BF	Static Analysis		Occurrences		Final Lazart Analysis		
		IP	Time	IP	Limit	AP	EP	Time
✓	-	104	1s	104 → 40	1	10	602	10s

Deps: Dependencies
BF: Brute Force (selection heuristic)
IP: Injection Points
AP: Attack Paths
EP: Explored Paths

Table 4: Results of the analysis of WooKey's iso7816 (vulnerable version, single data faults)

Deps	BF	Assertions		Static Analysis		Strategy Growing		Final Lazart Analysis		
		Total	Unproven	IP	Time	IP	Time	AP	EP	Time
-	-	-	-	-	-	660 → 655	16min	1	192	19s
✓	-	384	3	153	3s	153 → 151	6min	1	170	12s
✓	✓	384	3	1	55s	-	-	1	45	1s

Deps: Dependencies
BF: Brute Force (selection heuristic)
IP: Injection Points
AP: Attack Paths
EP: Explored Paths

remaining properties and the attack as the relative simplicity of the properties in relation to the code allowed Eva to be very precise. This is illustrated by the fact that over 99% of the assertion could be eliminated.

For this example we again used the strategy growing method in order to obtain results within a controlled time frame. Testing injection points based on occurrences did not help in this case as all of them were very similar. The resulting analysis times showcase the importance of reducing the number of injection points to consider as much as possible, as each one had to be individually checked.

5.3.2 Bootloader

The attack that was found originally on WooKey's bootloader used a single 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.

Table 5: Results of the analysis of WooKey's bootloader (single faults)

(a) without limits on injection point occurrences

Deps	BF	Static Analysis		Strategy Growing		Final Lazart Analysis		
		IP	Time	IP	Time	AP	EP	Time
-	-	-	-	485 → 474	19min	12	40 246	53min
✓	-	226	2s	226 → 215	15min	12	38 526	45min
✓	✓	45	3min	45 → 41	5min	12	13 592	6min

(b) with limits on injection point occurrences

Deps	BF	Static Analysis		Occurrences		Final Lazart Analysis		
		IP	Time	IP	Limit	AP	EP	Time
✓	-	226	2s	226 → 119	1	12	3 005	19s
✓	-	226	2s	226 → 129	10	12	4 518	36s
✓	-	226	2s	226 → 161	50	12	23 664	17min
✓	✓	45	3min	45 → 24	1	12	717	4s
✓	✓	45	3min	45 → 25	10	12	1 111	8s
✓	✓	45	3min	45 → 38	50	12	9 345	2.5min
✓	✓	45	3min	45 → 42	100	12	20 863	11min

Deps: Dependencies
BF: Brute Force (selection heuristic)
IP: Injection Points
AP: Attack Paths
EP: Explored Paths

Despite countermeasures being added consisting in doubling tests in critical functions, our analysis[7] shows that attack paths still exist as presented on Table 5.

Performance Discussion

As we were only interested in the "no firmware rollback" property[8] 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 full analysis being reduced by as much as 80% as shown in the first part of Table 5.

Interestingly, all of the dangerous injection points found only occur once. This is shown in the second part of Table 5 where all 12 attack paths are found regardless of the limit put on injection point occurrences. This means that all attack paths could be found in as low as 21 seconds[9], although there would be no indicator that there are no more at that point.

A few injection points were eliminated in every analysis, all of which are located within a countermeasure. We can thus assume that they have no impact alone.

Attack Paths Discussion

Our results allowed us to identify two attacks which look feasible in practice, both regrouping multiple possible paths, for a total of six attack paths among the twelve found. Other paths were dismissed due to being overly unrealistic, e.g. requiring to directly alter the boot function pointer, or serving no purpose to the attacker, e.g.

[7] The analyzed code is available as part of FISSC [10].

[8] "The most recent firmware is booted or an error / security breach is detected."

[9] With no brute-force and a occurrence limit of one.

allowing to switch to a special mode which can be trivially engaged by pressing on a button on the device.

The first attack consists in exploiting logic in the *loader_exec_req_selectbank* function, which decides which firmware to boot. Figure 12 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 well as that flop is not bootable in the next one, corresponding to the highlighted text on Figure 12.

The second attack takes advantage of a lack of countermeasures in the *loader_exec_req_flashlock* function, which computes the pointer to the boot function of the chosen firmware. Figure 13 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 with the highlighted text should be enough to solve this issue in a single fault context.

Verifying our Fixes

In order to ensure that our recommendations are valid, we applied our patch and tested it both in single and double fault contexts. As shown on Table 6 our fixes prevent the six previously identified dangerous attack paths in single-fault. However since all countermeasures on WooKey's bootloader are designed to protect in this context only, allowing a maximum of two faults should result in them being bypassed. Unfortunately multi-fault analysis is impractical in part due to the strength of our fault model. To illustrate this, we selected only injection points occurring once with projection and ran a strategy growing analysis on a cluster, which took around 30 hours to complete. Given that the final analysis is very incomplete, such runtime is unreasonable in most settings.

One way to solve this issue is to restrict our fault model to test inversion faults only. This drastically reduces the complexity of the analysis while inducing a loss of attack paths, although we can assume that this drawback is limited as all but two of the attack paths found in single-fault correspond to test inversions. As shown on Table 6 this analysis still yields a large amount of double-fault attack paths while being roughly 20 times faster compared to the cluster analysis. However most of these paths include a fault on the same injection point, *fault_82*, used in the four single-fault attack paths, thus they are redundant. If we remove *fault_82* we see that only six attack paths remain, which correspond to those we patched in single fault and thus our countermeasures can be bypassed in two faults.

5.4 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 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.

We also designed our method around the fact that our fault model does not allow for control-flow violations, which makes more sense at source level. Although, our method could be used with other fault models allowing for localized control-flow violations such as chaining *then* and *else* blocks after conditional jumps or skipping function calls.

Finally, multi-fault analysis remains difficult when many injection points cannot be eliminated but can be done with some concessions as we showed with our analysis of WooKey's bootloader.

5.5 Using our Method to analyze Cryptographic Implementations

Our method is focused on the analysis of large programs with many complex feasible execution paths rather than cryptographic code. We already discussed how these two analysis targets fundamentally differ, however we reckon that the idea of applying the main principles of our method in a cryptographic context should be explored.

The main hurdle with analyzing cryptographic code with our method is that the principles of confusion and diffusion in effet in this context tend

```
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          && flip_shared_vars.fw.bootable != FW_BOOTABLE) {
9          if(!(flop_shared_vars.fw.bootable == FW_BOOTABLE
10             && flip_shared_vars.fw.bootable != FW_BOOTABLE))
11             goto err;
12         ctx.boot_flop = sectrue;
13         //...
14     }
15     if (flip_shared_vars.fw.bootable == FW_BOOTABLE
16         && flop_shared_vars.fw.bootable != FW_BOOTABLE){
17         ctx.boot_flip = sectrue;
18         //...
19     }
20     //...
21 }
```

Fig. 12: loader_exec_req_selectbank function in WooKey's bootloader [9] (with ***fixes***)

```
1  static loader_request_t loader_exec_req_flashlock(loader_state_t nextstate){
2      //...
3      else if (ctx.dfu_mode == secfalse) {
4          if (ctx.boot_flip == sectrue) {
5              if (ctx.boot_flip != sectrue)
6                  goto err;
7              //...
8              ctx.next_stage = (app_entry_t)(FW1_START);
9          }
10     //...
11     }
12 }
```

Fig. 13: loader_exec_req_flashlock function in WooKey's bootloader [9] (with ***fixes***)

to render any kind of dependency analysis pointless, as everything should depend on everything else. As a result the selection of injection points based on dependency becomes equivalent to naive systematic selection. In practice, we were unable to eliminate any injection points on neither implementations of CRT-RSA nor AES from the FISSC benchmark [4].

Arguably, we find that the idea of limiting the number of injection points through static analysis to improve scalability is not as helpful for analyzing cryptographic implementations. Indeed such programs tend to be smaller and to adhere to well-known schemes, meaning that it is easier for experts to decide where faults should be injected in this case. Beating human judgement in this matter would therefore be more difficult with an automated analysis. Additionally the scalability issues encountered by tools such as Lazart in a cryptographic context are not only related to the number of possible execution paths but also to the complexity of the path constraints. We also note that Lazart can analyze the CRT-RSA implementation from FISSC without any issues using the state-of-the-art "set data to zero" fault model, including with the Aumuller and Shamir countermeasures [34], thus optimization is not always needed. It is only when attempting analysis with unconstrained symbolic faults that difficulties arise.

Table 6: Results of the analysis of Wookey's bootloader with fixes

Fault Model	Fault Number	Static Analysis		Strategy Growing		Final Lazart Analysis		
		IP	Time	IP	Time	AP	EP	Time
Full	1	230	2s	230 → 219	15min	6	38 548	46min
Full	2	145[a]	2s	145 → 142[b]	30h	735	113 559	19min
Test Inversion	1	77	2s	77 → 75	3.5min	4	11 724	4min
Test Inversion	2	77	2s	77 → 60	1.5h	274	25 517	2min
Test Inversion	2	77	2s	77 → 59[c]	-	6	24 907	2min

IP: Injection Points
AP: Attack Paths
EP: Explored Paths
[a]only injection points occurring once with projection
[b]done on a cluster
[c]without *fault_82*

6 Related Works

6.1 Static Analysis

In Christofi et al. [21] 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 [20], 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 2.5. 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.

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.

6.2 High-level Fault Analysis Tools

Larsson et al. [30] first proposed to simulate faults in Java applications by injecting symbolic bits at specified memory addresses. Their implementation involves manually instrumenting the code to indicate where injection points should be placed. This method is fundamentally similar to Lazart, with a focus on fault tolerance.

SymPLFIED [32] uses a single symbolic variable to propagate the effects of fault injection using propagation rules and model checking to find attacks. Contrary to Lazart, 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 the same scalability issues as symbolic execution.

ProFIPy [22] simulates faults in python programs according to a user-defined fault model by matching code patterns and replacing them with faulty ones. Mutants are thus generated for further testing. This work also focuses on fault tolerance but our method could still be used with it.

Le et al. [31] use symbolic execution at LLVM IR level to find faulty execution paths in C programs, with faults being simulated by intrucing symbolic bits. This is the closest available alternative to Lazart, although it is mainly designed to assert fault tolerance.

None of the previously mentionned tools support multi-fault analysis. Lazart is to our knowledge the only available state-of-the-art option for the purpose of high-level fault analysis of C programs with a focus on fault injection attacks and the ability to find attack paths with multiple simulated faults. Regardless, we argue that the previously discussed tools are similar enough to be compatible with implementations of our method in their respective languages.

7 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. Issues with non-terminating symbolic execution and multi-fault analysis should also be addressed further. 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 a hybrid approach could be adopted 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://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/. FISSC test suite, accessed may 2022.

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

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

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:199–224, 2019.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Profipy: Programmable software fault injection as-a-service. *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 364–372, 2020.

[23] 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.

[24] Aakash Gangolli, Qusay H. Mahmoud, and Akramul Azim. A systematic review of fault injection attacks on iot systems. *Electronics*, 11(13), 2022.

[25] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 258–269, New York, NY, USA, 2011. Association for Computing Machinery.

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

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

[28] 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.

[29] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS - 19th European Symposium on Research in Computer Security*, volume 8713 of *Lecture Notes in Computer Science*, pages 200–218, Wroclaw, Poland, September 2014. Springer

International Publishing.

[30] Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In *VERIFY*, 2007.

[31] Hoang M. Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 845–850, 2018.

[32] 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.

[33] 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.

[34] Maxime Puys, Lionel Rivière, Julien Bringer, and Thanh-Ha Le. High-level simulation for multiple fault injection evaluation. In *DPM/SETOP/QASA*, 2014.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.