Detecting fault injection vulnerabilities in binaries with symbolic execution

Julien Lancia

Detecting fault injection vulnerabilities in binaries with symbolic execution

Julien Lancia
NXP
Toulouse, France
julien.lancia@nxp.com

2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI) | 978-1-6654-9535-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ECAI54874.2022.9847500

***Abstract*—We propose a framework based on symbolic execution to identify automatically and consistently the effects of fault injection on embedded software at assembly level, for C (compiled to assembly) and Java (compiled to bytecode) binaries, for faults affecting the control flow or the values in memory. We implement our framework on top of the angr symbolic execution engine, with built-in support for various fault models (Stuck-at, Hamming weight, Unconstrained). We assess the performances of our framework on open source programs considering single and double fault injections, showing that it identifies all possible fault injections in a fraction of the time required by manual review.**

***Keywords*—security, fault injection, symbolic execution, embedded software**

I. Introduction

We observe that physical attacks on embedded software are becoming increasingly popular [1], [2] while at the same time the equipment to perform such attacks is becoming more and more accessible and affordable [3]–[5]. To protect against these attacks, founders and developers can implement hardenings both at hardware and software level. This is even more prominent for products targeting certifications such as Common Criteria with certification laboratories inspecting the source code and compiled binaries for vulnerabilities and running penetration testing campaigns on the products.

To address this challenge, the traditional approach consists of performing manual (static) source code security reviews to identify the potential vulnerabilities that could be exploited with hardware fault injection and adding hardening where necessary. However, given the large size of source codebases in today's embedded software, such a task may be time consuming, resource consuming and error prone.

```
1  x = int(input())
2  if x >= 10:
3    if x < 100:
4        print "ROM verified"
5        return 0
6    else:
7        print "Error abort 1"
8        return 1
9  else:
10   print " Error abort 2"
11   return 2
```

Listing 1. Pseudo-code example

For example, consider the pseudo-code in listing 1. Depending on the fault model considered the following fault injection could lead to an invalid status of ROM verified:

- one fault injected at line 2 can corrupt the conditional instruction and let the control flow progress to the ROM verified state at line 4,
- one fault injected on the x variable assignment at line 1 can change the value of x to satisfy the conditions and reach the ROM verified state at line 4,
- one fault injected at line 8 can corrupt the return statement and let the control flow progress after the return statement.
- one fault injected at line 11 can corrupt the return statement and let the control flow progress after the return statement.

We can see that a very small code extract has already several potential vulnerabilities, and we could potentially identify more if reviewing the compiled binary code instead of the source code. This highlights the need for a more automated and consistent way for security reviews of embedded software to protect against fault injections.

We propose a framework based on symbolic execution to identify automatically and consistently the effects of fault injection on embedded software at assembly level, both for C (compiled to assembly) and Java (compiled to bytecode) programs. Thanks to this flexibility, our framework allows detecting fault injection vulnerabilities in C binaries, Java applications, JavaCard applications, pure Java Android applications as well as native Android applications using a mix of C and Java through JNI calls.

Our symbolic execution framework performs code execution at symbolic level, stepping through all control flow blocks without concretizing the data. For each block, it generates both control flow fault injections and memory fault injections at symbolic level, keeping track of the faults that are generated along the symbolic execution. The fault model used for memory fault injection is fully customizable, only limited by the expressivity of the constraint solver. The identification of successful fault injection (indication of vulnerability) is also fully customizable and can be expressed by any condition on a symbolic state (expressed in python language in the current implementation).

Our symbolic execution framework identifies automatically all attack paths resulting from fault injections on the control flow or the memory. It can perform the analysis for an arbitrary number of fault injections in the same run, only constrained by the computing resources necessary to store and process the

resulting symbolic graph. In the following sections we first describe the principle of symbolic execution with a constraint solver, then we describe the symbolic fault engine added to the symbolic execution framework, and finally we present the current implementation of the framework.

The structure of this paper is as follows. Section II presents the generic concept of symbolic execution and our conceptual approach to simulate control flow and memory value fault injection at symbolic level. Section III overviews the implementation and tools used to implement our approach. Section IV assesses the performance of our approach and implementation on open source softwares in C and Java. Section V discusses related work. Section VI concludes this paper.

II. Symbolic Fault Injection

A. Symbolic execution

A symbolic execution engine executes the code at symbolic level, which means that the actual values the program is operating on (register, memory values) are replaced with symbolic placeholders that can take any value. Along the program execution, concrete operations on the actual values are modelized with constraints on the symbolic values.

When the program control flow branches, the symbolic execution engine virtually follows all branches and applies on each path a set of condition called path guards. Usually, a symbolic execution engine operates on an intermediate representation of the source or binary code which allows the execution engine to be independent of the language executed.

For example, given the source code presented in listing 1 the symbolic execution engine will generate the graph presented in figure 1.

In this graph, for convenience, the state number identifies the line number of the basic block (the code between two conditional or unconditional jumps) in the original source code the graph is representing. As a result, in this notation several states can have the same identifier, but in the framework each state is modelled independently.

We can see that the symbolic variable x is never concretized, however new constraints are added to the symbolic variable x when new conditional instructions are encountered. As the execution progresses, new symbolic variables are created, and constraints are added. The symbolic execution engine is coupled with a constraint solver where all the constraints are stored and associated with the relevant states. The constraint solver allows reasoning on the constraints at a given state. For example, on state 7 a request to the constraint solver "x $<$ 10" would return false because the constraints on the state don't allow it. This ability to perform symbolic analysis on states through the constraint solver becomes very handy when working on hundreds or thousands of constraints.

Input values that are not explicitly concretized are initialized as symbolic values (like the variable x in the listing 1). Therefore the symbolic execution engine explores all possible code paths without having to specify any specific input test value.

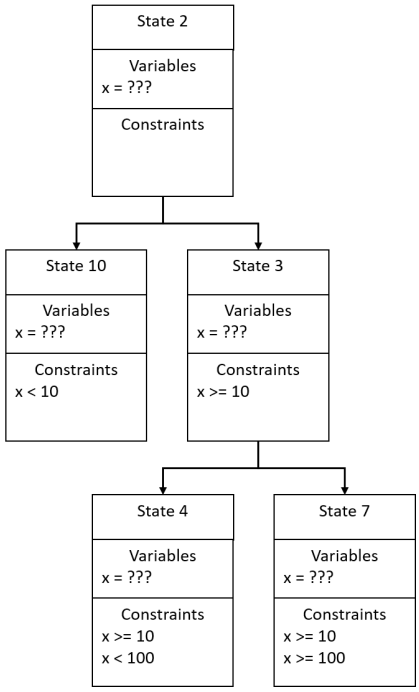

Fig. 1. Graph representation of the code

B. Symbolic fault engine

The symbolic fault engine is the core of our contribution. We developed as symbolic fault engine on top of a symbolic execution engine, thus leveraging the benefits of symbolic execution (systematic code exploration using symbolic input values, constraint solving) to identify fault injection vulnerabilities in binary executables. Our symbolic fault engine offers two main features: control flow symbolic fault injection and memory symbolic fault injection.

1) Control flow symbolic fault injection: Control flow symbolic fault injection mimics a hardware fault injection whose effect modifies the control flow of the program. At symbolic level, this control flow modification is represented by an additional state, where the guard constraint (the constraint that is added based on the entry condition) is inverted. In addition, the state is flagged as "faulted" to allow tracing the fault. The faulted state is equivalent to the original state, except the entry condition is inverted. The symbolic execution is carried out on the faulted state to allow identifying successful attacks in the remaining code, and to perform additional faults in the rest of the code.

We present in figure 2 a graph where a symbolic fault has been injected on state 3, creating a state 3F. We can see that the guard constraint "x $\geq$ 10" has been inverted to model the fault and is now "x $<$ 10", and the state is flagged as faulted. The execution graph continues from this state and the following constraints are added normally. In this example the faulted flow reaches the state 4.

A full representation of the faulted graph would contain a faulted state for each normal state to model all possible control flow faults on each state. To prevent combinatorial explosion,

```
x = 4
if x >= 10:
  if x < 100:
      print "ROM verified"
      return 0
  else:
      print "Error abort 1"
      return 1
else:
  print " Error abort 2"
  return 2
```

Listing 2. Pseudo-code example for value fault

we use the constraint solver to identify impossible states (states where two conditions are contradictory) and remove them from the graph. For example, the state 7 after fault injection has two contradictory conditions "x $<$ 10" and "x $\geq$ 10" so it would be automatically removed from the graph as unsolvable.

The faulted graph shows that the state 4 "ROM verified" can be reached with a single fault injection on control flow at state 3. If the success condition was expressed as "reaching instruction print ROM verified" the symbolic fault engine would store the path as a valid attack path.

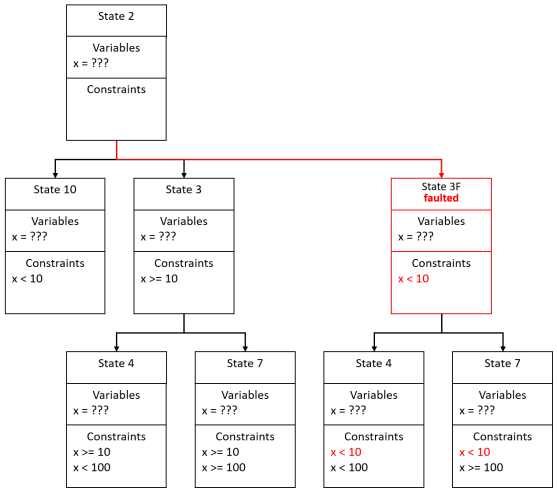

Fig. 2. Graph with control flow faulted state

2) Memory symbolic fault injection: Memory symbolic fault injection mimics a hardware fault injection whose effect modifies the value stored or read in memory. At symbolic level, this memory modification is represented by an additional state, where the constraint representing the memory update is modified according to a fault model. The memory symbolic fault injection is enforced on instructions that perform either memory write, or memory read.

We consider in listing 2 a modified version of the program presented in listing 1. In this new code, the x variable is assigned before checking its value.

The source code in listing 2 will be represented in the symbolic execution engine by the graph presented in figure 3. Given the value assigned to x, the line 4 (i.e. state 4) is identified by the solver as an impossible state, meaning it will never be reached in a normal execution. Indeed, the variable x cannot be simultaneously equal to 4 and greater equal to 10.

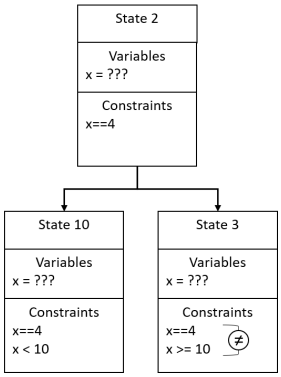

Fig. 3. Graph representation of the code for memory fault

Let us now consider a "stuck at" fault model. This fault model sets the value at a given address in memory to either all binaries 0 or all binaries 1. The memory symbolic fault injection would change the constraint in state 2 from "x==4" to "x==0 AND x==0xFFFF". The ability of the symbolic fault injection engine to reason on constraints rather than on concrete values allows capturing two different faults (resulting from the same fault model) in a single state. However, this fault model does not allow reaching the success state at line 4 as the path guard ("x $\geq$ 10 AND x $<$ 100") is not fulfilled by the faulted constraint.

Let us now consider a "bit flip" fault model. This fault model switches a single bit in memory. According to this fault model, the fault injection engine would now change the constraint in state 2 to "x==0 OR x==5 OR x==6 OR x==12 OR ... OR x==32772" (a constraint with 16 clauses, which can be easily generated with a bit shift). Here again, the symbolic fault engine compacts 32 different faults in a single state thanks to the use of the constraint model. The faulted constraint now allows reaching the success state at line 4, therefore the fault will be stored as a valid attack path. Figure 4 represents the graph resulting from the memory symbolic fault injection on state 2.

As shown in figure 4, the symbolic fault injection engine creates a new state for the memory fault injection, that will have its own execution flow after the fault injection.

When considering multiple faults, the number of vulnerabilities found is artificially increased. Indeed, one fault can be actively corrupting the binary execution flow to reach the success condition while the other faults are not relevant to the success condition. The framework reports these cases as vulnerabilities as the success condition is reached. A task of manual review of reported vulnerabilities is then necessary to sort the vulnerabilities that are single faults duplicates, and real double faults in the reported vulnerabilities. Note that running a first pass with only single faults enabled helps to eliminate the single faults and identify the successful double faults.

Figure 5 illustrates this particular case where a double fault duplicates a single fault. First fault on state 1 has no effect on

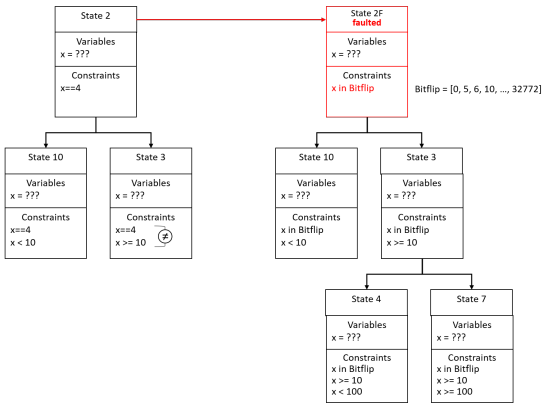

Fig. 4. Graph with value faulted state

```
1  project = angr.Project("./test_binary",
       auto_load_libs=False)
2  main = project.loader.find_symbol("main")
3  initial_state = project.factory.entry_state(entry=
       main)
4  sm = project.factory.simulation_manager(
       initial_state)
5  sm.use_technique(
6    Faulter(enable_cond_fault=True,
7    enable_write_fault=True,
8    max_faults=2,
9    mem_range=[main_addr, main_endaddr],
10 ))
11 sm.explore(find= 0x40116f)
12 for attack in sm.found:
13     attack.print_details()
```

Listing 3. Angr script with fault exploration technique

reaching the success condition as both faulted and unfaulted flows reach state 3. Only the fault on state 3 is relevant to reach the success state.

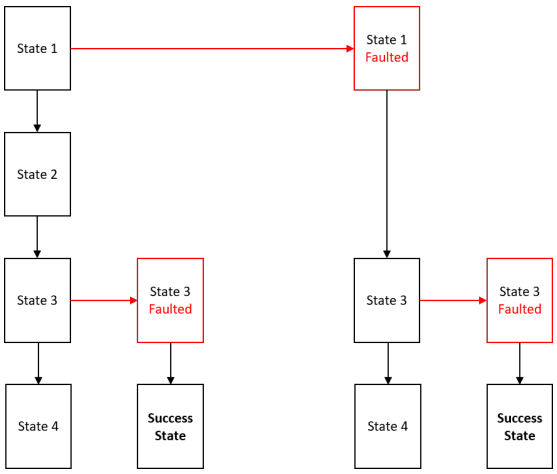

Fig. 5. Graph representation of a double fault duplicating a single fault

Both control flow and memory fault injection engine can be combined to allow representing a very large scope of the possible fault injections on an embedded application. The ability to represent fault models as constraints instead of values prevents combinatorial explosion of states even when representing complex fault models such as bitflip on 32-bit or 64-bit values. The resulting analysis provides a complete view of all possible attack paths given a set of success conditions.

III. Implementation

Our implementation relies on the angr symbolic execution framework [6]–[9]. This framework is composed of the following parts:

- CLE, the binary loader. It loads binary and creates an object-oriented representation of the binary. It supports several architectures and languages,
- Archinfo, the architecture database. Archinfo is a list of architecture description containing map of the register file, bit width, usual endian-ness, etc. CLE relies on this database to make binary representation,
- SimEngine, the symbolic execution engine,
- PyVEX, the intermediate language. It is actually a python wrapper of VEX, the intermediate representation used by the Valgrind tools. VEX is an architecture-agnostic, side-effects-free representation of various target machine languages.
- Claripy, the solver engine. Claripy is responsible for creating, composing, and eventually solving the symbolic constraints,
- SimOS, the implementation of OS level features (files, networking, I/O, ...). It offers a symbolic representation of OS resources to the the symbolic execution engine.

Angr offers several ways of extending the framework. Among these extension points, we use two to create our fault engines: state plugins and exploration techniques. State plugin allows adding properties to the states. We create a new state plugin to store all the information related to fault injection: faulted tag, history of previous faults injected, number of faults so far.

Exploration techniques define the behavior of the symbolic execution engines when it steps through the states. The default behavior is a ”step everything at once” strategy (effectively breadth-first search), but angr offers other exploration techniques such as depth-first search or thread-level parallelism. We define a new exploration technique that extends the default breadth-first search with control flow and memory fault injection capabilities. Concretely, when evaluating a new state, the exploration technique clones the state, updates the fault properties in the state fault plugin and updates the state constraints according to the fault model.

Listing 3 shows an angr script that loads a binary, applies the fault exploration technique and runs until the success condition is reached. The success condition is here defined as reaching a specific success address, but it could be any condition on the state.

At line 1 and 2 the binary is loaded and initialized to start at function main. At line 5 the simulation manager (symbolic execution engine) is configured to use our Faulter exploration

```
1 def check_memory(current_state):
2     if current_state.mem[0x41414141].long.concrete
          == 0:
3         return True
4     return False
5 sm.explore(find= check_memory)
```

Listing 4. Angr script with arbitrary fault success condition

technique. We can see that our Faulter exploration technique can be customized to enable or disable fault on control flow, on memory, to set the maximum number of faults on a path and to limit the address range the memory faults apply to. On line 11 the simulation manager is started, with the find parameter that defines the address to reach for success. This is the default success condition, but it is also possible to pass a function in the find argument. In this case the find function accepts a state argument and should return True in case of success. Listing 4 shows an example of a success condition defined as a function of state. Instead of defining the success condition as reaching a specific address, the check_memory method checks for a specific value at a memory address.

IV. Evaluation

Our evaluation is performed on a core i7 with 8GB of RAM. The framework runs on a Linux OS inside a VMWare virtual machine. Performance results are presented in table I. Fault model is either conditional or memory fault, depending on the type of fault chosen for the test. For memory faults, the fault model is either:

- Hamming-weight (HW): the faulted value takes all values with a hamming distance of 1 to the original value (using constraints on a symbolic variable),
- Stuck-at (SA): the faulted value is all 0s and all 1s (using constraints on a symbolic variable),
- Unconstrained(U) : the faulted value takes all values fitting in the variable type (using an unconstrained symbolic variable).

The initialization time is the time in seconds to load the binary in the framework. The run time is the time in seconds to execute the binary in the framework and explore all possible faults for the fault type and fault model chosen. The full time is the sum of the initialization time and run time. Finally, the number of vulnerabilities found is the number of faults that triggered the success condition, reported at the end of the execution.

For double faults test cases, the number of vulnerabilities found is artificially increased. Indeed, as explained in section II-B2, some double faults can be duplicating single fault effects.

Table I shows the time necessary for our framework to identify all fault injection vulnerabilities in test programs and real world, open source programs. From our experience, the analysis time of our framework represent a small fraction of what would be necessary to identify these vulnerabilities, including double fault injections, by manual review.

To evaluate our framework, we run a set of experiments and measure the performances. For each test, we evaluate separately:

- The conditional fault injection detection and the memory fault injection detection,
- For memory fault injection detection, the Hamming-weight, the unconstrained and the stuck-at fault models,
- Single fault injections and double fault injections.

A. Considerations on path explosion

During our experiment, we did not encounter path explosion issues for the program that we evaluated. However, it is possible that path explosion occurs when analyzing larger programs, which would cause the framework to freeze or crash. Indeed, angr default exploration strategy is a breadth-first strategy, meaning that it "steps everything at once" for a given program state. This exploration strategy is efficient but resource intensive. In case of path explosion issue, angr provides another built-in strategy that can be configured at instantiation using angr constructor parameters, Depth First Search, that keeps only one state active at once, putting the rest in a deferred state until it dead-ends or errors. This strategy explores one path at a time, thus trading resource for analysis time. Another approach when facing path explosion issues is to reduce the search space by limiting the analysis to the security critical functions of the program. This can also easily be done at instantiation using angr constructor parameters that define an address range or an accept-list of functions to be considered for symbolic execution.

B. Assessment on sample programs

We first evaluate the performances of our framework on small C and Java example binaries that we developed for this purpose.

1) We compile the small C program presented in listing 5 and run fault injection vulnerability detection on the resulting binary. The C program contains an unreachable statement at line 8. The success condition is defined as reaching this statement address using fault injection.
2) We compile the small Java program presented in listing 6 and run fault injection vulnerability detections on the resulting binary. The Java program contains two unreachable calls to the *success* method at line 12 and 17. The success condition is defined as reaching these method calls using fault injection.

C. Assessment on open source programs

After evaluating our framework on small C and Java example binaries, we then evaluate the performances of our framework on real-world C and Java binaries provided as part of open source projects.

1) openCryptoki: openCryptoki [10] is an implementation of the PKCS#11 API [11] that allows interfacing to devices (such as a smart card, smart disk, HSM) that hold cryptographic information and perform cryptographic functions. The openCryptoki API provides a standard programming interface

TABLE I
PERFORMANCE EVALUATION

Name	Fault type	Fault model	Nb. faults	Init. time (s)	Run time (s)	Full time (s)	Nb vuln found
C sample	conditional	-	1	0.07	0.55	0.62	1
		-	2	0.08	0.55	0.63	1
	memory	HW	1	0.14	1.10	1.24	2
		HW	2	0.06	3.01	3.07	10
		SA	1	0.17	1.24	1.42	0
		SA	2	0.07	2.35	2.42	0
		U	1	0.08	0.87	0.94	2
		U	2	0.07	2.18	2.25	11
Java sample	conditional	-	1	6.88	0.06	6.93	2
		-	2	6.67	0.03	6.71	2
	memory	HW	1	6.79	0.56	7.35	2
		HW	2	6.83	3.10	9.92	21
		SA	1	7.68	0.57	8.25	0
		SA	2	7.80	2.61	10.41	0
		U	1	8.07	0.97	9.04	2
		U	2	7.87	1.29	9.17	21
openCryptoki	conditional	-	1	0.13	5.66	5.79	2
		-	2	0.13	13.15	13.28	14
	memory	HW	1	0.12	26.04	26.17	1
		HW	2	0.14	103.40	103.54	14
		SA	1	0.30	26.75	27.05	1
		SA	2	0.12	370.83	370.94	22
		U	1	0.16	25.85	26.01	1
		U	2	0.14	313.81	313.95	22
Wallet	conditional	-	1	14.00	2.40	16.40	1
		-	2	12.20	9.06	21.26	13
	memory	HW	1	12.17	13.71	25.87	1
		HW	2	12.15	421.15	433.30	63
		SA	1	20.36	68.48	88.85	1
		SA	2	14.34	268.26	282.60	63
		U	1	16.67	12.93	29.60	1
		U	2	14.33	290.02	304.35	63

```
1  int main(int argc, char **argv){
2    int a = 0;
3    int b = 0;
4
5    a = b;
6
7    if(a==2){
8      b=10;
9    }
10
11   return 0;
12 }
```

Listing 5. Simple C binary for performance evaluation

```
1  class TestJava {
2
3      public static void main (String[] args){
4          run();
5      }
6
7      public static void run(){
8        int a = 0;
9        int b = 0;
10
11       if(a==2){
12         b=10;
13         success();
14       }
15
16       if(b==2){
17         b=10;
18         success();
19       }
20     }
21
22     public static void success() {}
23 }
```

Listing 6. Simple Java binary for performance evaluation

between applications and all kinds of portable cryptographic devices. Certain PKCS#11 operations, such as accessing private keys, require a login using a Personal Identification Number, or PIN, before the operations can proceed.

We compile the sample login program provided by the openCryptoki project and run fault injection vulnerability detection on the resulting binary. Listing 7 shows an extract of this sample program. The success condition is defined as reaching the logged state at line 12 of the listing without providing a valid PIN to the program.

When running the program in our framework, we disable external library loading. As a result, all PKCS#11 API calls are hooked by the framework and automatically return symbolic values. This allows us to run the program without deploying the full PKCS#11 infrastructure such as the PKCS#11 demon and file structure.

2) Oracle JavaCard wallet: JavaCard is a Java-based OS dedicated to secure embedded platforms. JavaCard OS and

```
rc = funcs->C_OpenSession(slot_id, flags, NULL,
    NULL, &session);
if (rc != CKR_OK) {
    show_error("C_OpenSession", rc);
    return rc;
}
rc = funcs->C_Login(session, userType, (
    CK_CHAR_PTR) pass, strlen(pass));
if (rc != CKR_OK) {
    show_error("C_Login", rc);
    return rc;
}

printf("Logged in successfully, logging out...\n")
    ;

rc = funcs->C_Logout(session);
if (rc != CKR_OK) {
    show_error("C_Logout", rc);
    return rc;
}
```

Listing 7. openCryptoki program for performance evaluation

applications are often protected against physical fault injections and these protections are typically assessed when performing security evaluations such as Common Criteria. We thus decided to evaluate the performances of our framework on JavaCard based applications.

Oracle provides, as part of its Java Card Platform Development Kit [12], a Wallet sample to demonstrate a simple cash card application. It keeps a balance, and exercises some Java Card API features such as the use of a PIN to control access to the applet.

We compile the Wallet program provided by Oracle and run fault injection vulnerability detection on the resulting applet. Listing 8 shows an extract of this applet, where the PIN is checked in the debit method to allow the debit operation. The success condition is defined as reaching the debit operation at line 30 without providing a valid PIN to the applet.

Since the applet is a Java program and JavaCard API implementation is not provided as part of the Java Card Platform Development Kit, all Java Card API calls are automatically hooked by the framework and return symbolic values. We had to add a special hook for the ISOException.throwIt method to end the program when the API is called to get a consistent behaviour.

V. Related work

In [13]–[15] the authors use a similar approach based on symbolic execution, but their approach is focused on control flow fault injection while our framework supports both control flow fault injection and memory fault injection. Moreover, their approach is targeting C only while our framework supports both C and Java.

In [16] the authors focus only on memory fault injection. Their approach requires code annotation to specify the location of the fault and the targeted variable. On the contrary our framework support C and Java and does not require any code annotation as the analysis is performed on the whole code and faults are generated for every possible memory accesses.

```
private void debit(APDU apdu) {

    // access authentication
    if ( ! pin.isValidated() )
        ISOException.throwIt(
            SW_PIN_VERIFICATION_REQUIRED);

    byte[] buffer = apdu.getBuffer();

    byte numBytes =
        (byte)(buffer[ISO7816.OFFSET_LC]);

    byte byteRead =
        (byte)(apdu.setIncomingAndReceive());

    if ( ( numBytes != 1 ) || (byteRead != 1) )
       ISOException.throwIt(ISO7816.
           SW_WRONG_LENGTH);

    // get debit amount
    byte debitAmount = buffer[ISO7816.OFFSET_CDATA
        ];

    // check debit amount
    if ( ( debitAmount > MAX_TRANSACTION_AMOUNT)
         ||  ( debitAmount < 0 ) )
       ISOException.throwIt(
           SW_INVALID_TRANSACTION_AMOUNT);

    // check the new balance
    if ( (short)( balance - debitAmount ) < (short
        )0 )
         ISOException.throwIt(SW_NEGATIVE_BALANCE)
             ;

    balance = (short) (balance - debitAmount);

} // end of debit method
```

Listing 8. Wallet sample program for performance evaluation

In [17] the authors work on a first order fault model targeting C program control flow to skip instructions (using NOP or JUMP instructions). In contrast, our framework supports arbitrary number of fault injection, support both C and Java and support both control flow and memory fault injection.

[18], [19] are tools based on concolic testing. Concolic testing is a variant of symbolic execution approach where symbolic execution runs simultaneously with concrete executions. Generally, this approach is used to reduce combinatorial complexity of symbolic execution: concrete execution is used to test the program, and symbolic execution is used along the concrete execution path to determine the input that would raise coverage.

In [18] the authors use concolic execution to generate functional unit testing of Java programs. In [19] the authors use concolic execution to generate functional tests of C binaries.

In [13]–[15], the authors use the KLEE concolic execution engine to identify fault injections targeting control flow modifications that allow reaching a specific point in code. As presented above, this work differs from ours as our framework supports both control flow fault injection and memory fault injection. Moreover, their approach is targeting C only while our framework supports both C and Java. Finally, we can

identify fault success with any symbolic condition while they only identify fault success as reaching a specific address in code.

[20] proposes an automatic tool for IoT software, "Chaos Duck", to detect sensitive data leaks, program crashes and corruptions in control flow caused by fault injections. Chaos Duck modifies a disassembled binary to produce faulted binaries according to different fault models: modifying a branch instruction, setting an instruction to a nop or setting a variable to zero at variable declaration time. Compared to our approach, this work only covers a subset of fault models as we take into account any variable corruption happening at any time during the program execution. Morevover, this work targets only C binaries, while our framework supports both C and Java binaries.

In [21], the authors propose an approach based on model checking to simulate fault injection by generating mutant binaries. This approach differs from ours as it only considers fault models on control flow while our approach considers fault models on both control flow and memory values. Morevover, their implementation targets only C binaries, while our framework supports both C and Java binaries.

Concolic execution could be a possible improvement of our approach when combinatorial complexity becomes too high. Our prototype is implemented with the angr framework that supports concolic execution, so concolic execution could be added to our prototype without changing framework.

VI. Conclusion

We propose a framework based on symbolic execution to identify automatically and consistently the effects on fault injection on embedded software at assembly level. Our symbolic fault injection framework:

- supports C, assembly and Java, thus enabling vulnerablity detection in C binaries, Java applications, JavaCard applications, pure Java Android applications and native Android applications;
- supports both control flow and memory fault injection;
- automatically analyses the whole program for every possible control flow and memory faults;
- supports arbitrary fault model based on symbolic constraints;
- support arbitrary success condition.

To our knowledge, none of the current existing tools targeting fault injection vulnerability analysis on binary offer the same features.

References

[1] "Trustzone-m(eh): Breaking armv8-m's security," https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security, accessed: 2021-06-02.

[2] "There's a hole in your soc: Glitching the mediatek bootrom," https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom/, accessed: 2021-06-02.

[3] "Chipwhisperer," https://github.com/newaetech/chipwhisperer, accessed: 2021-06-02.

[4] "Chipshouter," https://github.com/newaetech/ChipSHOUTER, accessed: 2021-06-02.

[5] "Voltpillager," https://zt-chen.github.io/voltpillager/, accessed: 2021-06-02.

[6] "Angr," https://angr.io, accessed: 2021-06-02.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[9] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[10] "opencryptoki," https://github.com/opencryptoki/opencryptoki, accessed: 2021-06-02.

[11] O. Standard, "Pkcs# 11 cryptographic token interface base specification version 3.0," 2020.

[12] Oracle, *Java Card 3 Platform Development Kit User Guide*. Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065: Oracle, 2015, no. Version 3.0.5.

[13] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 213–222.

[14] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "Fissc: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 3–11.

[15] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, "Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks," in *International Symposium on Foundations and Practice of Security*. Springer, 2014, pp. 92–111.

[16] D. Larsson and R. Hähnle, "Symbolic fault injection," in *International Verification Workshop (VERIFY)*, vol. 259. Citeseer, 2007, pp. 85–103.

[17] X. Kauffmann-Tourkestansky, "Analyses sécuritaires de code de carte à puce sous attaques physiques simulées," Ph.D. dissertation, Université d'Orléans, 2012.

[18] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

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

[20] I. Zavalyshyn, T. Given-Wilson, A. Legay, R. Sadre, and E. Riviere, "Chaos duck: A tool for automatic iot software fault-tolerance analysis," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 46–55.

[21] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay, "An automated and scalable formal process for detecting fault injection vulnerabilities in binaries," vol. 31, no. 23. Wiley Online Library, 2019, p. e4794.