Reversing Obfuscated Code of a Java Card Through Code Reconstruction

Yasmina Berrou[1], Mohamed Benmohamed[2], Fatima Bourabaa[2], Billel Kenidra[2], Mohamed Badeche[2]

[1]Computer Science Department, University of Batna 2, Batna 05000, Algeria
[2]Laboratoire d'Informatique Répartie (LIRE), University of Constantine 2, Algeria
E-mail: yasminaberrou@yahoo.fr,
mohamed.benmohammed@univ-constantine2.dz,fatima.bourebaa@univ-constantine2.dz, b_kenidra@esi.dz,
mohamed.badeche@univ-constantine2.dz

Keywords: Java card security, reverse engineering, backtracking search algorithm

Received: April 1, 2026

In secure devices like Java Cards, several attacks have been developed to dump memory and reverse its contents, thereby gaining access to sensitive stored assets. Many manufacturers have proposed various countermeasures to protect this data from malicious code. Obfuscation is one of these techniques; it consists of converting part or all of the code into another form that is much harder to understand in order to make the reverse engineering of memory dumps more difficult. This conversion occurs instantly during downloading. In this paper, we propose an automatic approach to reverse engineer obfuscated intermediate bytecode based on a backtracking search algorithm. We then introduce heuristics to optimize the search process and accelerate convergence toward realistic solutions. We implement this approach in a prototype, and, to improve it, we integrate it into the Java Card Disassembler and Analyzer (JCDA) tool. We also explain the new tricks used by our tool to recover some secure assets. Our tool successfully reversed 92% *of illegal opcodes and all secure assets.*

Povzetek: Prispevek predstavi avtomatiziran pristop za povratno inženirstvo obfuskirane vmesne kode na Java Card napravah, ki z uporabo iskanja z vračanjem in heuristikami učinkovito rekonstruira kodo ter omogoča obnovitev zaščitenih podatkov.

1 Introduction

Java Card is a lightweight Java variant designed for resource-constrained secure components that store sensitive and secure assets such as PINs, cryptographic keys, and executable code. The Java Card platform is the most widely adopted technology in such embedded secure elements and IoT applications. It's open access, which enables and facilitates the post-issuance loading and execution of applets, making it a prime target for various physical, logical, or combined attacks aimed at dumping the card's memory and reverse-engineering it to recover secure data. To mitigate these attacks, the capability to download the code onto the secure device is ensured by a protocol presented by GlobalPlatform [4].

Countering these attacks is the primary objective of the smart card industry, which has implemented extensive measures and defenses to bolster device security. This includes providing guidelines [2], certification processes [5], and test suites [20] to ensure the secure design and runtime operation of such applications.

When someone succeeds in dumping a card's memory, several manufacturers increase the complexity of reverse engineering the card by hiding certain information using techniques like obscurity or a processor datasheet that is not publicly available.

Obscurity (or obfuscation) is an effective security technique that transforms code into obfuscated code. deliberately making it harder for humans or tools to read, analyze, or reverse engineer while preserving its semantics and functionality. This approach protects smart cards against fault attacks and combined (logical + fault) attacks. However, it can also be exploited to introduce malware into devices [11, 18], making it a double-edged sword.

In this paper, we describe the latest techniques of obfuscation to avoid the reverse of the dump in Java Card. Then we present an approach to reverse the obfuscated code. In other words, to find the original code by using the concept of state memory and a backtracking search algorithm. This research addresses the questions

RQ1 Can a backtracking search algorithm recover the semantics of obfuscated opcodes in partially encrypted Java Card memory dumps?

RQ2 How can we integrate this algorithm into an existing tool (e.g., JCDA), and what techniques enable recovery of secure assets?

The rest of the paper is organized as follows. In Section 2, we provide background on Java Card security and the most common obfuscation techniques used to mitigate reverse engineering of embedded code. In Section 3, we

introduce our approach to reversing obfuscated code by detailing the concept of memory state and the proposed backtracking search tree algorithm. In Section 4, we implement our approach in a prototype and, to enhance it, integrate it into the Java Card Disassembler and Analyzer tool. In Section 5, we present our experiments and explain how our tool retrieves secure data. In the final section, we sum up with a few projects for the future.

2 Background

2.1 Java card security

To be adapted to the limited device of Java Card, Java Card Standard Compiler produces as output a CLASS FILE that is transformed a more compact representation, the converted applet CAP FILE. Java Card embedded in a smart card transforms a CAP FILE into an executing applet.

A smart card's security depends on its built-in sensors (light, heat, voltage, etc.) and its countermeasures. These sensors protect the card against hardware attacks such as laser beams and side-channel attacks. They immediately erase the card's memory contents if such an attack is detected or disable the card to preserve confidentiality of sensitive secure containers (crypto keys, PIN counters, etc.).

Software countermeasures are another way of protecting the card; they can be integrated into the card to prevent shell code execution (viruses) during runtime or positioned externally to verify the application's syntax and semantics prior to loading onto the card.

ByteCode Verifier (BCV) checks whether the converted applet (CAP file) is correctly typed and verified if the security rules defined by Java Card specification [1] are respected. To prevent illegal access to memory. For example, with the Java language, it is impossible to execute the branch instruction with a primitive type; it must be a reference type. BCV checks only the static analysis, which is an important and necessary step before loading the applet into the Java Card. However, due to its high cost in terms of time and memory consumption, most of the Java smart cards do not support the integration of BCV on the card. Therefore, a trusted third party verifies the code off-card and signs the application. Then on the card, during loading and installation, the digital signature is ensured.

Moreover, the Java Card Firewall provides isolation between packages, performing a check at runtime to protect the applet from reading and writing data in an applet in another package. If multiple applet instances share the same package; they operate under identical security contexts. Each applet receives a distinct security context domain, matching that of the package responsible for its creation. Java Card Firewall separates contexts, preventing a method running in one context from accessing or invoking any method of an applet in a different context. The Shareable Interface Object (SIO) is the only way to share functionality between objects, which prevents an applet from exporting a set of methods through the firewall to protect the management of card content (loading, installation, deletion, customization, etc.). The security policies of the Global Platform Card Specification [1] secure messaging protocols and integrity/confidentiality mechanisms.

There are other countermeasures embedded in the card, in addition to the one defined by the specification. For example, a typed stack [22] may be utilized where a primitive type (such as short) is inserted at the top of the stack, with references below, or vice versa. This arrangement helps to prevent confusion-type attacks. Additionally, some cards include checks on jump boundaries and other similar measures.

2.2 Memory carving of java card

Even with all the security mechanisms supported by Java Card virtual machine, many attacks have been developed in the literature to dump the NVM area and then reverse the dump. We do not present all these attacks here, but we concentrate on related papers that offer tricks.

2.2.1 Dumping NVM

NVM (Non-Volatile Memory) is utilized to save applications installed after the card's issuance. To dump it. There are multiple ways to do this:

The first one changes the parameter of unchecked static instructions. The aim of the attack, presented by [14], is to allow unrestricted read/write access across all memory through unchecked instructions (`getstatic`, `putstatic`, and `invokestatic`). The argument of these instructions is the reference (token or address). If we change these arguments, we can access everywhere in memory. This attack has been developed in an old card and can be mitigated by various countermeasures. Authors in [25] present two attacks to dump the memory. The first attack modifies a local variable's index to retrieve and write the current method's return address with the array's address, causing the array to execute when the method finishes. This array contains shellcode that allows dumping the `NVM`. The second attack modifies the operand of the bytecode instruction `jsr` to update the return address to the array's address, then executes the shellcode.

Exploiting platform implementation flaws is another way to dump memory. Examples of these include flaws in the off-card bytecode verifier [23] or flaws in the transaction mechanism implementation [17], whereby the shareable mechanism is abused due to non-typed checking [8].

Another method involves altering the array's metadata, or treating the data as metadata, utilizing the idea of reference forgery. The array's metadata includes the type, security context, and size; if the array's size is increased, the entire Non-Virtual Memory (NVM) regions may be dumped. A revisited version of the attack [15] is presented in [13]. This version of the attack uses the Java Card API's `ArrayCopyNonAtomic` method to trick the system into thinking that the metadata is an array header. The

outcome of this attack is a large dump of the NVM with a length of `0xFFFF`, which allows access to memory areas that were inaccessible with the forges. The authors of [9] and [10] introduced a novel attack known as "Auto-Forge". It is predicated on searching through a series of data in memory and compels the Java Card Virtual Machine (JCVM) to accept them as legitimate metadata for an array. It is possible to dump the memory after the set sequences have been located. It can then simply write new data, which can be regarded as metadata, to the memory to dump and read more memory pieces. In [6] present several attacks to dump the memory and retrieve secure assets.

The authors of [18] recently repurposed the notion of a fault-enabled virus. They employed the idea of construction code to impose on and deceive the system into perceiving an application that is not ill-typed as a well-typed application, and they activated after loading into the card through a laser beam attack. The authors can access all of the NVM memory thanks to this behavior. The Phi-attack is developed in [21]; it is a type of attack that exploits type confusion through shared interfaces and modified export files `(.exp)` in Java Card applets.

2.2.2 Reversing the dump

Reverse engineering, also called backward engineering, is the process of extracting information from the dump and reproducing it at a higher level of abstraction. In Java Card memory, the reverse engineering is to obtain a source code from binary code; it can be split into two separate phases:

System level reversing Is used to determine the general structure and functionality of the system, then distinguish between the binary representation of data and code. In Java Card memory, the loading format is known as the Converted APplet (CAP) file. It includes eleven components: Header, Directory, Applet, Class, Import, Method, Constant Pool, Static Field, reference location, Descriptor, and Export. Each component details a specific part of the CAP file's contents, like class data, executable bytecode, linking details, verification info, and more. Identifying each package and examining it is essential to understand their implementation and interconnections.

In the studies presented in [6, 10], pattern-matching analysis is used to extract all Java-based data. This analysis identifies all stored class data and examines how the system manages it. First, it identifies all Java Card objects and then attempts to link these objects together. Finally, it infers how the system manages the dynamic components.

Code level reversing Extracting a control flow graph (CFG) is a difficult task because it requires analyzing binary code to identify the sequence of bytecode instructions in a program. This process examines the code in a low-level format and translates it into a higher-level representation. Several tools are available for performing static binary analysis and extracting structural data from binary code, including disassemblers, binary translators, decompilers, and binary rewriters.

This study focuses on disassemblers, which are used to derive the assembly-language representation of binary code extracted from a dump. Among commercial tools, **IDA Pro** is one of the most widely used and offers capabilities such as control-flow analysis and function identification. Reverse engineering has become increasingly difficult because older cards have a direct correspondence between internal binary representations and external assembly-language instructions. The bytecodes used in these cards conform to the Java Card specification [1]. The authors of the paper [26] provide an automatic tool that converts binary code to bytecode instructions based on the index of coincidence.

However, on recent cards, encrypted memory can be found, where a single instruction may be encoded by different binary codes depending on the expected context. The challenge is to reverse a sequence of binary code without knowing the instruction set of the target processor. Mesbah et al. in [5] present an approach for inferring the semantics of unknown instructions based on the principles of a bytecode verifier (BCV). This approach attempts to identify an abstract type for each unknown instruction and subsequently infer its semantics. Kasmi et al. in [19] show that Java Card bytecode obfuscation can be bypassed using reverse engineering aided by side-channel analysis and demonstrate practical recovery of illegal opcode behavior.

2.2.3 Obfuscation technique to mitigate reverse engineering memory dump

There are several ways to prevent reverse engineering of Java Card memory. In this work, we focus on the following two techniques:

Code compression Compressing multiple instructions into new instructions results in a more compact and more complex program, which makes the reverse-engineering of memory dumps more difficult. This method requires a compressor to be built into the card in order to bypass bytecode verification. The authors in [12, 24] propose expressing new instructions as macros over existing instructions, with **JCVM** extended only to support general macro-instructions. This method allows programs to be accepted with or without compression. To analyze new instructions, authors in [25] propose an alternative method that uses global macros instead of traditional macros. Their idea is to put the macro in the ROM area and use the unused opcodes as shortcuts to common patterns. In the work by Massimiliano and Wolfgang [12], the authors focus on compressing the method component by combining two techniques. The first technique is based on a folding mechanism, which replaces sequences of Java bytecodes with equivalent single macro-instructions that extend the instruction set of the virtual machine. The second technique uses dictionaries,

replacing repeated sequences of bytecodes with macros defined in a dictionary.

Instruction encryption Instruction encryption consists of encrypting part or all of the bytecode instructions during installation. Barbu et al. in [27] introduce a countermeasure designed to hinder reverse engineering of memory dumps. Their approach scrambles each instruction during installation so that the code is verified by **BCV** (Byte Code Verification) before being loaded onto the card. A randomly generated key is then used to conceal both the instruction and its operand, as expressed by

$$inst_{\text{new}} = inst_{original} \oplus \text{key}$$

To reverse the dump, the attacker should be discovering the XOR key. So, to find the key, it should just change the control flow graph (CFG) from the program to a return instruction. In fact, the opcode of the return instruction is `0x7A`, which is one byte, so there are 256 possible values for the coding process. It can easily be found by brute-forcing it. To address the weaknesses of the previously described approach, Razafindralambo et al. [8] propose a revised method that utilizes XOR with a dynamic element, specifically the Java Program Counter (`JPC`). This approach is detailed in the following function:

$$inst_{\text{new}} = inst_{\text{original}} \oplus \text{key} \oplus JPC$$

In this case, each instruction has different opcode values depending on the context.

3 Proposed approach: auto-reverse of illegal opcode

Java Card bytecode can only use opcodes from `0x00` to `0xB8` for standard operations, plus two other opcodes, `0xFE` and `0xFF`, for specific use. Most Java Card applications contain additional opcodes outside of this scope, referred to as "illegal opcodes" which correspond to unknown or unspecified instructions in the Java Card virtual machine. In Listing 1, a fragment of dumped Java Card memory is presented, representing a 78- byte segment of non-volatile memory (NVM) starting from the logical address `0x11F0`. This memory fragment is analyzed to determine the presence of any illegal opcodes. Our study indicates that 29.25% of the bytes in this fragment contain illegal opcodes.

We propose below an automatic and generic approach to reverse illegal opcodes of intermediate bytecode instruction, using the concept of memory state and a backtracking search algorithm.

3.1 Concept and assumption

The idea of our approach is to find a sequence of instructions that connects two code fragments: the first, before the execution of the illegal opcode, and the second, after their execution, while respecting Java Card constraints; it is based on the concept of memory state, which is defined in [18, 20] as consisting of local variables and the operand stack, where each instruction is viewed as a relation between two concrete memory states: the memory state before instruction execution and the memory state after instruction execution.

```
/*0x11F0*/  00 0C 18 00 0A 78 65 6C
/*0x11F8*/  6C 6F 57 6F 74 6C 64 00
/*0x1200*/  00 02 80 71 00 03 04 10
/*0x1208*/  0C 34 00 00 01 BE 81 08
/*0x1210*/  00 0A 02 19 00 25 00 01
...
/*0x1258*/  2E 00 01 0D 48 65 6C 6C
/*0x1250*/  6F 57 6F 62 6C 45 41 70
/*0x1268*/  70 01 72 00 00 32 04 00
/*0x1260*/  05 07 03 00 00 01 73 01
/*0x1278*/  75 00 25 32 28 8D 19 98
/*0x1270*/  18 00 19 06 18 01 87 07
/*0x1288*/  18 08 BA BC 08 18 65 00
```

Listing 1: fragment of memory dump

Our method involves inferring the memory states at the end of code fragment 1 (initial state) and at the beginning of code fragment 2 (target state). Next, using a backtracking search tree algorithm, we attempt to identify one or more instructions that can map the initial state to the target state (Fig. 1). The detected instruction sequence reveals the specifications of the illegal opcodes.

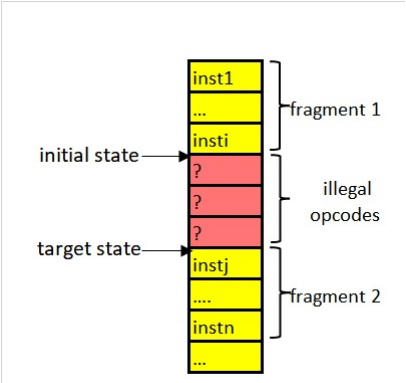

Figure 1: Auto reverse concept

Our framework begins by introducing the concept of memory state and then infers the initial state and the target state in Section 3.2. Then, using a backtracking search tree, we attempt to identify one or more paths, or solutions, that can map the initial state to the target state, as described in Section 3.3. The instruction-selection algorithm must adhere to Java Card constraints, meaning that structural and typing properties must be satisfied, and we then try to optimize it. Finally, in Section 3.4, we select the best solution if more than one is found. The detected instruction sequence

identifies the specifications of illegal opcodes.

To solve those challenges, our approach is based on the following assumptions:

1. The target is a Java Card memory dump.
2. The memory is partially encrypted (the code contains a legal opcode and an illegal opcode).
3. This approach works on the techniques defined in Section 2.2.3.
4. The reverse engineering is done at the byte code level and in the same direction as the CFG.

3.2 Memory state

3.2.1 Formalization

Each memory state M_i is formally defined as a pair: $M_i(S_i, V_i)$, where:

- S_i is the operand stack.
- V_i are the local variables.

For the operand stack, we used abstract execution at the type level rather than on concrete values to examine raw data. The execution of the instruction is performed at the level of types, not on specific values as in normal execution. The number of operands stack does not exceed the `Max-stack` defined in the header of the method. Primitive types include `byte`, `short`, and `boolean`, which are only supported by Java Card instructions. The local variables are stored in the registers, and their number and type are fixed at the header of the method during the execution of the method.

The relationship between successive memory states is defined by a set of Java Card byte code instructions `inst1 inst2 inst3` ... given in the Java Card instruction specification JCVM[1]. Where: For each `instk`

$$M_i(S_i, V_i) \xrightarrow{\texttt{instk}} M_{i+1}(S_{i+1}, V_{i+1})$$

3.2.2 Calculation

Using the semantics of each bytecode instruction to interpret the relationship between two memory states before and after execution of this instruction. Each semantic of the instruction is a formal specification of its functional effect on the operands stack, local variables, and control flow in the JCVM environment, ensuring safe and predictable execution in the secure, resource-constrained Java Card platform.

According to the JCVM specification, each instruction processes specific inputs and produces outputs to change the memory state (Table 1) through the following:

1. Consumes its inputs and produces outputs at the top of the stack in the new memory state.
2. Updating the list of local variables.

For example, the arithmetic instruction `sadd` modifies the top of the operands stack by consuming two short that are present at the top of the operand stack in the state, denoted M_0, and produce a short in state M_1 after execution. For example, the arithmetic instruction `sadd` modifies the top of the operands stack by consuming two short that are present at the top of the operand stack in the state, denoted M_0, and produce a short in state M_1 after execution. In symbolic form, this instruction can be represented as:

$$M_0([short, short], [none]) \xrightarrow{\texttt{sadd}} M_1([short], [none])$$

Register access instructions such as `sload` and `sload_n` generate an equality constraint (same type) between the top of the stack and the local variable indexed by n (Table 1); i.e., they push the type of local variable n onto the top of the stack.

$$M_0([none], [short]) \xrightarrow{\texttt{sload}} M_1([short], [short]);$$

3.3 Backtracking search tree algorithm

To determine all possible solutions (sets of instructions) from initial state to target state using a backtracking search tree, we propose a new search tree algorithm. It is based on depth-first search to generate and explore all memory states and two stopping criteria: when it reaches the target memory state or the maximal depth (MaxDepth) (random generation and fixed in advance). (Fig. 2).

In Fig. 2, each node in the tree contains an instruction and its associated state. For each node, we assign pointers: one to the parent node and others to its child nodes. We begin with the root (initial state and beginning instruction), choosing a subset of instructions, and for each chosen instruction, we compute the associated state (node) in level 1. Then, the algorithm recursively explores one path until the stop criteria are met (depth-first search) before backtracking to the parent node to explore another unvisited child. We repeat this process for each node. We backtrack when either the associated state is the target state (saving the path as a solution) or when the number of instructions in the path (level n) exceeds a predefined maximum depth (MaxDepth).

In Algorithm 1, starting from the initial state by creating a root node, a depth-first search function of the tree begins at the root, dives deeply along the leftmost path first, backtracks only upon meeting stop criteria, and eventually returns to the root. The algorithm functions by maintaining a path of instructions taken from the root. When a state transition leads to the target state, the algorithm saves the path. If a branch fails to reach it within the allowed depth or the target state is reached, it performs backtracking by removing the last instruction from the path and exploring the next instruction in the set of instructions. If no solution is found, a change of `Maxdepth` is necessary.

The limitation of the original approach is that it treats every possible next instruction (every node in the search tree)

Table 1: Example of memory state before and after instruction execution

Instruction	Memory state before instruction execution.	Memory state after instruction execution.
`sadd`	$M_0([short, short], [none])$	$M_1([short], [none])$
`aload_0`	$M_0([none], [ref])$	$M_1([ref], [ref])$
`aload_1`	$M_0([none], [none, ref])$	$M_1([ref], [none, ref])$
`bspush`	$M_0([none], [none])$	$M_1([short], [none])$
`getfieled`	$M_0([ref], [none])$	$M_1([short], [none])$
`astore_0`	$M_0([ref], [none])$	$M_1([ref], [ref])$
`sstore_1`	$M_0([short], [none])$	$M_1([short], [none, short])$

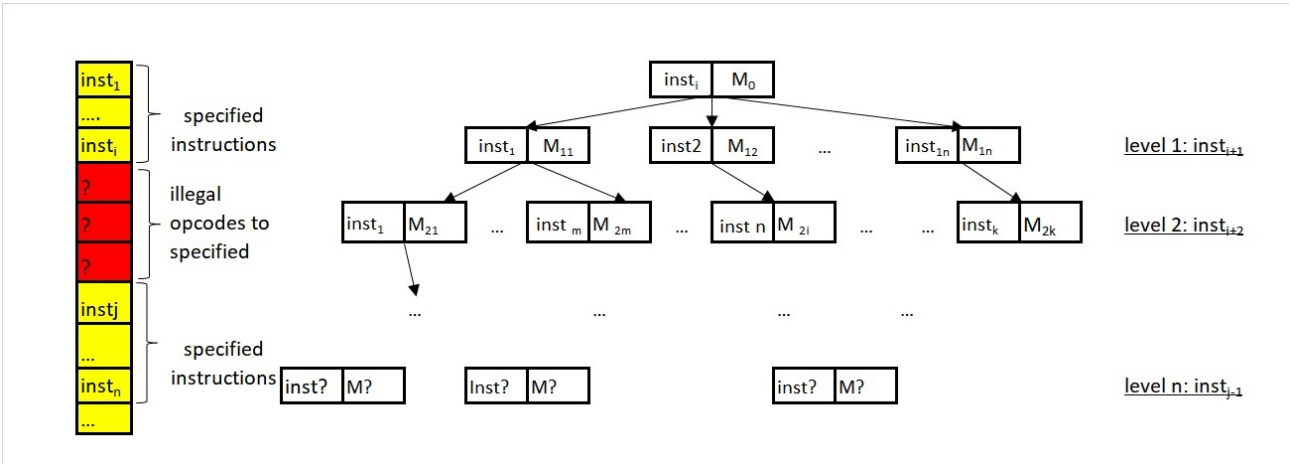

Figure 2: Backtracking search algorithm structure

Algorithm 1 Reverse ingeneering illegal opcode

```
1: Input initialState, beginInstruction, subset of instructions, targetState, MaxDepth.
2: Result possiblPath (a set of instructions capable of connecting the initial state (root) and the target state).
3: rootNode ← CreateNode(initialState, beginInstruction, null)
4: path ← [ ]
5: DFS(rootNode, 1, path)
6: function DFS(node, depth, path)
7: if node.state = targetState then
8:     Save(path)
9:     add path to possiblPath return
10: end if
11: if depth > MaxDepth then return
12: end if
13: for each inst in subset of instructions do
14:     associateState ← ComputeState(node.state, inst)
15:     child ← CreateNode(associateState, inst, node)
16:     path.push(inst) DFS(child, depth+1, path)
17:     path.pop()
18: end for
19: end function
20: end
```

in the same way, i.e., randomly selecting them with equal probabilities, which wastes effort exploring sequences that rarely, if ever, appear in real code and increases the algorithm's time complexity.

To enhance and optimize it, we created data files for each instruction, containing `trigram statistics`, where it predicts the most likely next instruction based on the current (second) and preceding (first) instructions, prioritizing high-frequency successors to guide tree search or prediction efficiently on the frequency of its following instruction. By analyzing bytecode from numerous applets, stats reveal that some instructions appear far more frequently than others. In practice, just 45 of the 250 standard Java Card instructions account for 90% of executed bytecode. Load operations (like `sload_1`) make up roughly $30 - 40\%$ of it. For a given trigram (prev → current → candidate), the algorithm loads the current node's stats file to select top-k successors first, pruning low-probability branches early. (Table 2)

3.4 An example

To illustrate the proposed approach, we present an example of applying algorithm 1 to known instructions, i.e., we treat legal opcodes as illegal opcodes (instructions between lines 4 and 8 in listing 2). The purpose is to see if we can locate those instructions in the set of solutions.

To simplify the graphical representation, we associate each memory state with its operand stack and the chosen

Table 2: Example of trigrams list for some instructions

Precedent instruction	Curent instruction	Instruction followed possible
`sload_1`	`sload_2`	`sadd sand aload_1 aload_2 dup ….`
`aload_1`	`getfieled_a_this`	`aaload checkcast dup goto …`
`sload_1`	`sadd`	`sload_1 aload_1 sload_3 sand …`
`sadd`	`sstore`	`sload_1 sload_2 sload_3 if …`
`sstore`	`dup`	`aload1 sload if ...`

```
1 02 // flags : 0 max_stack: 3
2 12 // nargs: 1 max_locals: 2
3 ...
4 0x10AA bspush 0xAA // [short]
5 0x31   sstore_2 // []
6 0x19   aload_1  // [ref]
7 0x03   sconst_0 // [short,ref]
8 0x02   sconst_1 // [short,short,ref]
9 ...
```

Listing 2: Illustration example

bytecode instruction.

As previously mentioned, to optimize and minimize the graph size, we selected a subset of instructions that can follow the last instruction in the path (Table 2), limiting it to only 3 or 4 instructions in this example. The process of execution is shown in Fig. 3.

Input:

- Beginning instruction `bspush`
- Initial Operand stack [short]
- Target Operand stack [short, ref]
- Local variables [short, ref]
- MaxStack = 3
- Maxdepth = 3

In (Fig. 3), we begin with the `bspush` instruction and the initial state; we choose four instructions that can follow it: `aload_1`, `sstore_0`, `sconst_2`, and `bspush`. Then we calculate the associated state of each following instruction. This process is repeated for each instruction and its associated state. We backtrack if the associated state matches the target state (as specified in the input) or if the depth exceeds 3. In this example, we discover two solutions. One of them (the path contains the colored states) is the same instruction set specified previously in Listing 2.

3.5 Choosing the best solution: integration and verification

To obtain the best solution, we insert each solution, i.e., instruction set generated by the algorithm, into the memory dump (instead of unknown instructions) using a CAP file manipulator, and we keep only the solutions that can be accepted by BCV. If multiple solutions are accepted, we have two cases:

1. Solutions do not have the same path length; we choose the solution with the fewest instructions, i.e., the shortest path from the starting state to the target state, due to embedded system constraints.
2. Solutions have the same path length; we choose the solution that appears first. (that contain the most frequent instructions)

This process ensures that the CAP file is formatted correctly and that it accurately reflects the appropriate arguments for certain instructions, such as jump instructions. (Fig. 4)

4 Ultimate java card disassembler and analyzer tool

To reverse illegal opcodes from Java Card code, we have developed a prototype tool to reverse automatically illegal opcodes inside the code. It takes as input an HTML page containing the reverse of all legal opcodes from the memory dump and produces a text file with the reverse of all illegal opcodes. To improve it, we will integrate it into the JCDA tool presented in [25], and we called it the Ultimate Java Card Disassembler and Analyzer (UJCDA) (Fig. 5).

JCDA, or the Java Card Disassembler and Analyzer, is a tool designed to automatically reverse Java Card memory dumps, where the code in the dumped memory contains only legal opcodes. It is implemented in Java and must be adapted to the architecture of each Java Card model. It utilizes a pattern matching algorithm and the index of coincidence to efficiently locate data zones and code zones, as well as to define bytecode and native code.

The UJCDA tool, based on the JCDA tool, is used to identify and locate data and applet codes. The Java Card Analyzer analyzes the Java Card memory dump and employs a pattern matching algorithm to differentiate the data area from the Java bytecode area. It then identifies various types of data, including package information, classes, and methods. Then once the code area is recognized, it utilizes a coincidence index to reconstruct the original CAP binary file.

Once the source code (in binary format) of the applet stored on the Java Card is obtained, the Java Card disassem-

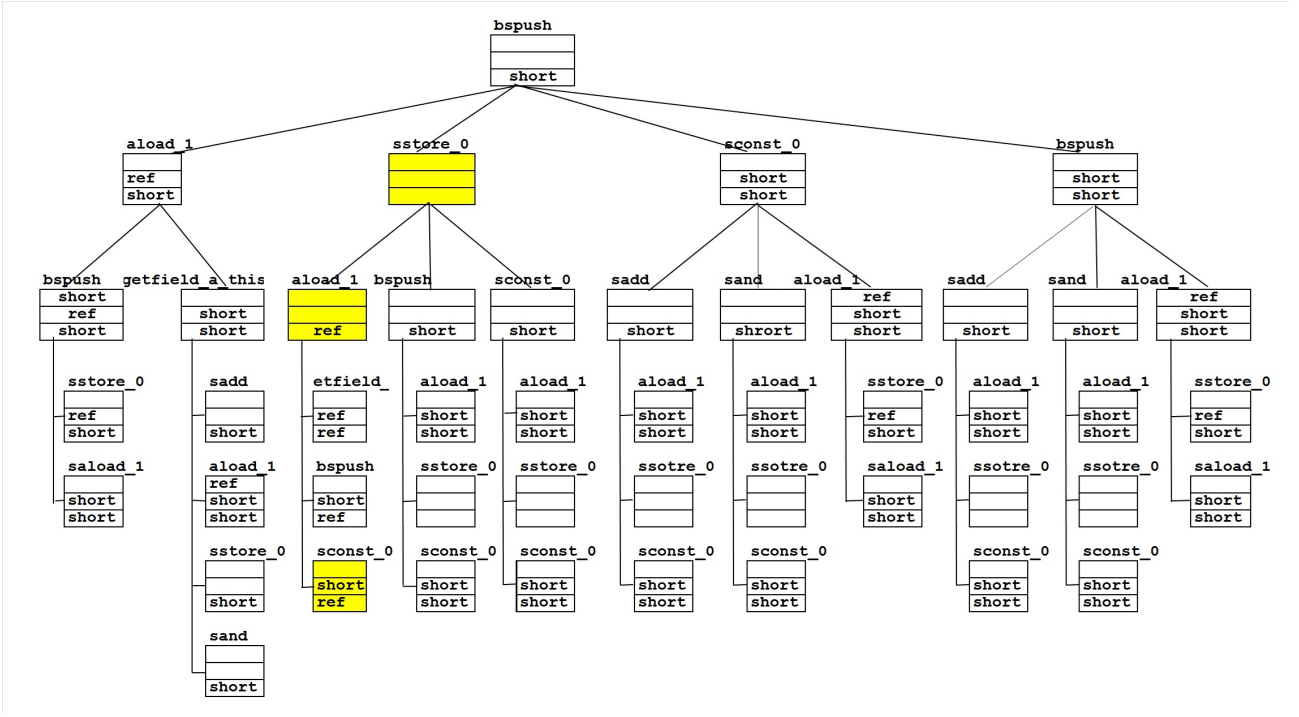

Figure 3: Illustration example

bler process begins. This process is divided into two parts: one for legal opcodes and another for illegal opcodes. The first part uses instruction specifications (including bytecode values, number of parameters, and sizes) (Table 3) to reverse legal opcodes. Then, it performed symbolic execution to extract the control flow graph (CFG) of each detected method as well as to determine the memory state for each instruction executed. This step is crucial to establish the memory state before and after executing illegal opcodes.

The second part of the disassembler process begins once all legal opcodes in the Java Card's dumped memory have been specified. This phase focuses on reversing illegal opcodes presented as unknown instructions. As previously described, an automated process based on a backtracking search algorithm is employed to reverse these illegal opcodes. This approach allows for the specification of the instruction represented by these opcodes, despite their obfuscation.

The result of our tool is an HTML file that reports tables, applet AIDs, and the reverse of the program's byte codes (the address of each legal and illegal opcode, along with their specifications and memory states), and also the reverse of sensitive data like PIN and RSA keys.

5 Experimental result

In this section, we describe our experiments designed to reverse engineer the memory dump of a Java Card 3.0.4 (which covers 90% of current commercial cards [3]). To retrieve secure assets like RSA private keys, PINs, and reverse native methods.

5.1 Tool output

We use the attack described in [21] to dump the memory. Listing 1 shows a fragment of this card's memory dump. Then we use our tool to reverse it. A portion of UJCDA output is shown in Table 4.

As shown in table 4, the resulting memory state is assigned to each instruction after its execution. An illegal opcode can be specified by one instruction, like `bspush` (its original opcode is `0x11`), or by more than one, as in the case of the illegal opcode `0xBB`.

For another experiment, we studied the behavior of both approaches defined by Mosbah's approach [6] and UJCDA on this card by increasing the size of the dumped memory. Fig. 6 shows the results of this experiment. The time required by Mosbah's approach increases dramatically on this card, as the tool requires an increase in the size of the dump to reverse all the unknown instructions. Since Mosbah's approach infers and reverses each unknown instruction separately, the time increases accordingly. On this card, the UJCDA approach performs better because it reverses all unknown instructions simultaneously.

5.2 Reverse native method

In the previous experiment, we have reversed the NVM dump. In this experiment, we used the knowledge gained from the reverse to explain how our tool reverses the contents of the ROM, which contains native methods. A native method is a method implemented in a language other than Java (such as C or machine code) that a Java Card can call. The `Method CAP file component` is used to de-

Table 3: Java card instruction specification

	Instruction	Parameter number	stack
0x1C	`sload_0`	0	$\rightarrow value1$
0x8D	`invokestatic_1`	(2byte parameter)	[arg] $\rightarrow$
0x41	`sadd`	0	value1 , value2$\rightarrow value$
0x87	`putfield_a`	(2byte parameter)	$\rightarrow value1$
0x77	`areturn`	0	value$\rightarrow empty$

Table 4: Ujcda output

Address	Instruction specification	Resulted state
0x64EC	0X04 //flags: 0, max_stack: 4	
0x64ED	0X12 //nargs:1, max_locals: 2	
0x64EE	0X10BB bspush 0xBB	[short][Ref,short]
0x64F0	0x31 sstore_1	[][Ref,short]
0x64F1	0x19 aload_0	[ref][Ref,short]
0x64F2	0x03 sconst_0	[short,ref][Ref,short]
0x64F3	0x02 sconst_m1	[short,short,ref][Ref,short]
0x64F4	0xCB14 bastore	
	getstatic_0x14	
	ssotre_0	[][Ref,short]
0x64F6	0X1C sload_1	[short][Ref,short]
0x64F7	0X19 aload_0	[ref,short][Ref,short]
...		

scribe methods that are defined within the CAP file. The `method_info.method_header_info` (Fig. 7) specifies whether the method is native or not, depending on the value of the flag item.

We use this item to characterize all native methods inside the card. And after a deep search, we found the reference of those methods inside the native methods table. The Java Card VM maintains an internal table mapping each native method token to its reference; at execution time, a specific instruction `0xBC` is used to call this method through its token (Fig. 8).

Reverse engineering Java Card native methods requires locating the native method table in ROM by pattern-matching to a small array that contains index sequences and pointer references (Fig. 9).

Once this table is identified, it provides direct entry points to all native methods. These are then reversed by rewriting them in the `8051` assembly language (Listing 3), with IDA Pro integrated into the JCDA tool.

```
...
0xb134    push 1
0xb135    push 0
0xb136    mov a, @0x02
0xb138    subb a, #0x00
0xb13a    mov  r0, r2
0xb13b    mov  r1, r3
0xb13c    mov  r2, #ff
0xb13d    mov  r3, #ff
0xb13f    mov dptr, #0x0292
0xb141    mov a, r2
0xb142    movx @dptr, a
0xb144    inc dptr
0xb145    mov a, r3
...
}
```

Listing 3: Snippet of native method reverse

5.3 Retrieve RSA private key

Java Card stores RSA private keys as `RSAPrivateKey` (modulus/exponent form) objects in EEPROM. An RSA private key instance must contain several fields that we have to retrieve: type, length, modulo, and exponent arrays (listing 4). The method to retrieve the `RSAPrivateKey` is to extract the RSA modulus n and private exponent d from Java Card memory dumps. Then, using

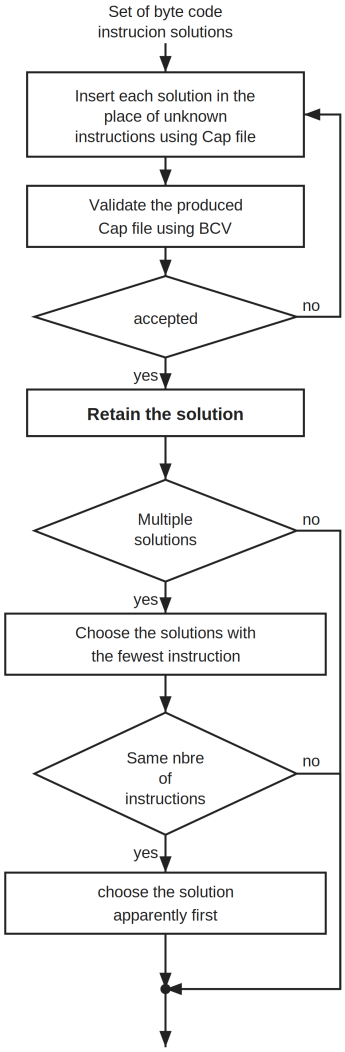

Figure 4: Verification and selection process

`javacard.security. KeyBuilder` to reconstruct it.

Listing 4: RSA private key structure

```
RSA_private_key structure {
u2 reference to exponent array
u2 reference to modulos array
u1 key length
u1 key type
}
```

Java Card uses the same type for all secure containers. They are difficult to distinguish from each other (PIN, DES key, RSA private key, etc.), so obtaining the characteristic properties of each secure container is necessary for identification and retrieval. In the case of an RSA private key, these properties derive from its components (modulus and exponent) leveraging their fixed sizes and big-endian format.

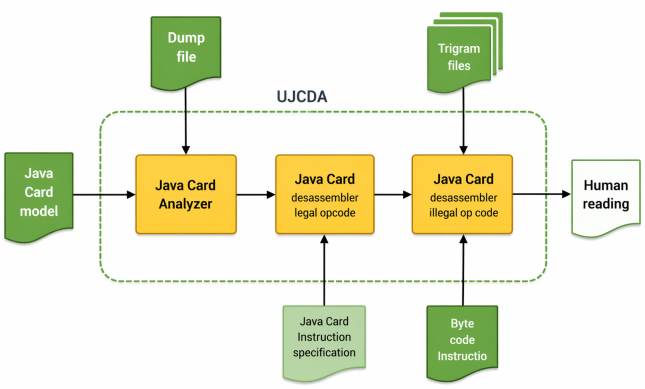

Figure 5: Architecture of ujcda

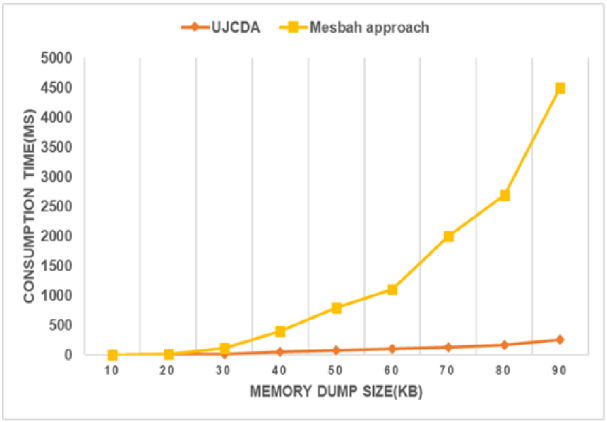

Figure 6: Time consumption of ujcda and mosbah approach

5.4 Retrieving PIN

Java Card `OwnerPIN` objects implement PIN functionality through the `javacard.framework. OwnerPIN` class, storing the PIN value, try limit, try counter, max size, and validated flag in a secure container typically marked by a specific type. (Fig. 10)

The idea is to dump NVM twice the first before a PIN attempt without incrementing the try counter. Once you have submitted a test PIN via APDU, we dump the NVM again. This differential method reveals transient PIN bytes during `check()` execution. Successful verification flips the validated flag from 0 to 1 without try counter increment; mismatches update the try counter value. Comparison of the two dumps precisely locates the array referencing the PIN value.

6 Conclusion and future work

In this paper, we present a generic tool to reverse engineer the memory of secure devices, especially Java Card-based smart cards. The purpose is to retrieve sensitive data and

```
Method_header_info {
u1 bitfield{
bit[4]flags
bit[4] max stack
}
u1 bitfield{
bit[4]nargs
bit[4]maxlocals
}
}
```

Figure 7: Native method header info

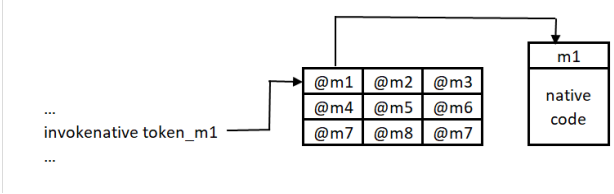

Figure 8: Native method call process

native code and to demonstrate the weaknesses of certain security techniques, such as obfuscation, by reversing obfuscated code. We propose an automatic approach to reverse unknown instructions of intermediate language based on the backtracking search algorithm. This method utilizes the concept of variable state memory, which allows for the calculation of memory states before and after the execution of unknown instruction bytecode. By employing the backtracking search algorithm, we can explore all possible solutions effectively. The proposed approach passes through three steps: first, we use the JCDA tool to infer the state memory before (starting) and after (destination) the execution of unknown code; next, we attempt to identify every path that could lead from the starting state to the destination state; and last, if there are numerous paths, we select the shortest one. Finally, we implement it in a prototype and then integrate it into the JCDA tool described in [25] in order to enhance it. This approach is usable even if the specification of the illegal opcode changes depending on the context of the method. In future work, we aim to extend this methodology to additional secure devices, such as uSIM, also known as Universal Subscriber Identity Module, which is a critical component in 3G, 4G, and 5G mobile networks, serving as an advanced application on the Universal Integrated Circuit Card (UICC) that stores subscriber identity and handles authentication. While maintaining the core principles of our approach and considering the specific requirements of this application.

Token	method name	adress
0x311D	readByteRam()	0x1EA5
0x312D	writeByteRam()	0x1F8F
0x321D	readSByteVMSTACK()	0x1702
0x322D	writeByteVMSTACK()	0x171B
0x331D	readShortRam()	0x17BB
…		
0x6161	APDU.getBuffer()	0x5701
0x6162	APDU.setOutgoing()	0x5A18
0x6163	APDU.setOutgoingLength()	0XD71F
…		
0xE61A	Util.arrayCopy()	0x2916
0xE61B	Util.arrayCopyNonAtomic()	0x7B16
0xE61C	Util.arrayFillNonAtomic()	0xBC16
…		

Figure 9: Native method table

```
OwnerPIN structure {
u2 reference to PIN value
u2 reference to transient array of the
flag
u2 try counter
u2 Max size
u2 PIN size
}
```

Figure 10: OwnerPIN structure

References

[1] Oracle (2023), Java Card™ Platform, Virtual Machine Specification, Classic Edition, ed., vol. 3.0.5 Classic, O. America, Ed.

[2] J. Gemalto (2006), Java Card and STK Applet Development Guidelines, Version 2.0, Gemalto.

[3] Zaoral, L., Dufka, A., Svenda, P. (2024). The Adoption Rate of JavaCard Features by Certified Products and Open-Source Projects. In: Bhasin, S., Roche, T. (eds) Smart Card Research and Advanced Applications. CARDIS 2023. Lecture Notes in Computer Science, vol 14530. Springer, Cham. https://doi.org/10.1007/978-3-031-54409-5_9

[4] Oracle (2012), Java Card System - Open Configuration Protection Profile, Version 3.0, Oracle.

[5] EMVCo, LLC (2013) EMV Testing and Certification White Paper, Version 1.0, EMVCo, LLC. [Online]. Available: http://www.emvconnection.com/downloads/2013/05/EM-Testing-Certification-V1.0-080213.pdf

[6] A. Mesbah, J.-L. Lanet, and M. Mezghiche (2019) Reverse engineering Java Card and vulnerability exploitation: a shortcut to ROM, *International Journal of Information Security*, vol. 18, no. 3, pp. 367-385. DOI:https://doi.org/10.1007/s10207-018-0401-9

[7] S. Volokitin and E. Poll (2016), Logical attacks on secured containers of the Java Card platform, in Smart Card Research and Advanced Applications, Berlin.

[8] T. Razafindralambo, G. Bouffard, B. Thampi and J.L. Lanet (2012), A dynamic syntax interpretation for Java-based smart cards to mitigate logical attacks, in International Conference on Security in Computer Networks and Distributed Systems, Berlin, Allemagne. DOI:https://doi.org/10.1007/978-3-642-35416-0-13

[9] A. Mosbah, L. Regnaud, j. L. Lanet and M. Mezguiche,(2016), The hell forgery, polymorphic codes shoot again, in 15th Smart Card Research and Advanced Application Conference.

[10] A. Mesbah, J.-L. Lanet et M. Mezghiche (2017), Reverse engineering a Java Card memory management algorithm, Computers and Security, vol. 66, pp. 97-114. DOI: https://doi.org/10.1016/j.cose.2017.01.005

[11] A. A. Azeta, A. Awal, J. I. Azeta, S. L. Hamunyela and A. d. Santos (2024) "Designing a Code Obfuscation Scheme for Software Protection," International Conference on Electrical and Computer Engineering Researches (ICECER), Gaborone, Botswana, pp. 1-5, Doi:https://10.1109/ICECER62944.2024.10920373

[12] Z. Massimiliano and R. Wolfgang (2014), A lightweight compression method for Java Card technology, in EWiLi'14, Lisbon, Portugal.

[13] Farhadi, M., Lanet, JL. (2017), Chronicle of a Java Card death, J Comput Virol Hack Tech 13, 109–123 (2017). DOI: https://doi.org/10.1007/s11416-016-0276-0

[14] J. Lancia (2012), "Java Card combined attacks with localization-agnostic fault injection," in International Conference on Smart Card Research and Advanced Applications.

[15] J. Iguchi-Cartigny, J., Lanet, JL. (2010) "Developing a Trojan applets in a smart card", J Comput Virol 6, 343–351 (2010). DOI: https://doi.org/10.1007/s11416-009-0135-3

[16] IDApro,"HexRays",[Online].Available: http://www.datarescue.com/idabase/overview.htm

[17] E. Hubbers, E. Poll (2004), Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviors, University of Nijmegen.

[18] S. Hamadouche, J. L. Lanet and M. Mezguiche (2020), Hiding a fault-enabled virus through code construction, Journal of Computer Virology and Hacking Techniques. DOI:https: doi10.1007/s11416-019-00340-z

[19] Kasmi, M.A., Azizi, M., Lanet, J.-L.: Reversing bytecode of obfus-cated java based smart card using side chanel analysis. Int J SecurAppl 9(11), 347–356 (2015). http://dx.doi.org/10.14257/ijsia.2016.10.2.32

[20] C. Florence and G. Arnaud (2010), "Constraint-based test input generation for Java bytecode, in 21st International Symposium on Software Reliability Engineering (ISSRE).

[21] J. Dubreuil and G. Bouffard (2021) "PhiAttack: Rewriting the Java Card Class Hierarchy," in Proceedings of the 20th International Conference on Smart Card Research and Advanced Application Design (CARDIS 2021), Lübeck, Germany.

[22] E. Faugeron (2011), Manipulating the frame information with an underflow attack, in International Conference on Smart Card Research and Advanced Applications.

[23] Léopold Ouairy, Hélène Le Bouder, and Jean-Louis Lanet. 2020. Confiance: detecting vulnerabilities in Java Card applets. In Proceedings of the 15th International Conference on Availability, Reliability, and Security (ARES '20). Association for Computing Machinery, New York, NY, USA, Article 24, 1–10. https://doi.org/10.1145/3407023.3407031

[24] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller (May 2000), Java bytecode compression for low-end embedded systems, ACM Trans. Program, Lang. Syst. 22, 3 (May 2000), 471–489. DOI: https://doi.org/10.1145/353926.353933

[25] Bouffard, G., Iguchi-Cartigny, J., Lanet, JL. (2011). Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (eds) Smart Card Research and Advanced Applications. CARDIS 2011. Lecture Notes in Computer Science, vol 7079. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-27257-8_18

[26] Massimiliano Zilli, Wolfgang Raschke, Johannes Loinig, Reinhold Weiss, and Christian Steger. 2013.

On dictionary compression for Java Card environment. In Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES '13). Association for Computing Machinery, New York, NY, USA, 68–76. https://doi.org/10.1145/2463596.2463605

[27] G. Bauffard and J. L. Lanet (2014), Reversing the operating system of a Java-based smart card, Comput. Virol. Hack. Tech, vol. 4, no. 239–253, p. 10.https://doi.org/10.1007/s11416-014-0218-7

[28] G. Barbu (2012), On the security of Java Card platforms against hardware attacks, Cryptography and Security [cs.CR]. Telecom ParisTech.

[29] Bouffard, G. (2025). Contributions to the security of embedded software in the chain of trust [Habilitation à diriger des recherches (HDR)]. ANSSI, Laboratoire d'architectures matérielles et logicielles.