JCAlgTest: Robust identification metadata for certified smartcards

Petr Svenda, Rudolf Kvasnovsky, Imrich Nagy, Antonin Dufka
Masaryk Univesity, Brno, Czech Republic
svenda@fi.muni.cz

Keywords: Certification, Smartcard, JavaCard, Cryptography, Forensics

Abstract: The certification of cryptographic smartcards under the Common Criteria or NIST FIPS140-2 is a well-established process, during which an evaluation facility validates the manufacturer's claims and issues a product certificate. The tested card is usually identified by its name, type, ATR, and Card Production Life Cycle (CPLC) data. While sufficient to pair the purchased card to its original certificate when bought from a trustworthy seller, such static metadata stored on the card can easily be manipulated. We extend the currently used card identification with a more descriptive set of metadata extracted from supported functionality, performance profiling, and properties of generated cryptographic keys. All of this information can be obtained directly by the evaluation facility, appended to the certificate, and later verified by the end-user with no need for any special knowledge or equipment, resulting in a better assurance about the purchased product. We developed a suite of open tools for the extraction of such characteristics and collected results for a set of more than 100 different smartcards. The database, openly available, demonstrates the significant variability in the measured properties and allows us to estimate the trends in support of different cryptographic algorithms as provided by the JavaCard platform.

1 Introduction

To facilitate the manufacturing of more trustworthy products and to remove the burden of verifying claims made from the product's end-user, security certification schemes like Common Criteria (Common Criteria, 2017) or NIST FIPS 140-2 (NIST, 2014) were introduced. A product is submitted for independent assessment by an accredited evaluation facility under a relevant scheme. The evaluation facility issues the security certificate after evaluating the claims made about product security features and the whole product lifecycle. The certificate is then publicly available to potential customers and may be considered or required for their purchasing decision.

Multiple issues became evident over more than two decades of operation of the certification schemes like Common Criteria. It proved to be time-consuming, costly, and not agile enough to keep pace with the product evolution after the initial certification was granted. Additionally, the crucial details of the evaluation are often not publicly available due to non-disclosure agreements protecting the intellectual property of both manufacturers as well as the evaluation facility and usage of closed tools. Finally, the binding between the certificate product and its certificate is typically loose. The product usually somewhat evolved due to natural development like the inclusion of new functionality or the application of security patches. The evaluation facility typically states not only the name and type of the product but also other identifying information like ROM mask versions or source code tags in a version control system. However, buyers are severely limited in their options to verify these identifiers as some reported by a device can be easily changed (e.g., version of the product), and some are not even available (e.g., source code for referenced tags). As a result, the buyer has to trust the seller that the device matches the claimed certificate. Such a situation becomes a problem when third-party resellers have to be considered – a typical situation for cryptographic smartcards when bought in smaller quantities or second-hand purchases.

We envision an improved certification procedure where an evaluation facility uses well-defined processes and open tools to analyze the claimed security of the product. Together with the exact settings used and artifacts obtained, the results are published along with the standard security certificate. The buyers can later repeat the relevant parts of the performed testing themselves or contract a specialized laboratory.

A suite of open-source tools was developed and

described in this paper. The suite can be used for the analysis of smartcards with the JavaCard platform, including other usage scenarios like an audit of the batch of cards purchased, forensic analysis of unknown or malfunctioning cards, and supporting developers to select a card with desired supported algorithms and performance. While the proposed approach might be possible to extend to some of the other types of certified devices, the smartcard domain itself already constitutes the most populous type certified under the Common Criteria scheme[1].

Contributions
This paper makes the following contributions:

- **Smartcard Audit Framework:** We introduce a systematic methodology for JavaCard benchmarking, which comprehensively evaluates the capabilities and the performance of the examined card. The primary usage is to verify the similarity of two or more cards. (Section 2)
- **Ecosystem Study:** We evaluate a wide variety of cards from all major smartcard manufacturers for the supported cryptographic algorithms, their performance, and information included in the Card Production Life Cycle, making this the first such large-scale study of the JavaCard ecosystem. (Section 3)

All presented tools and collected results are publicly available, including their source code, allowing for independent usage and verification by all involved parties.

2 JCAlgTest testing suite

In this section, we outline the functional modules of our testing suite called JCAlgTest, which provides a means to comprehensively gather, evaluate, and compare the capabilities of cards with the JavaCard platform (JC). The suite is split into several modules based on the type of gathered data. The modules consist of a host application (written either in Java or Python) and a small JavaCard applet uploaded on a card. The collected data is stored as raw text files, later processed by analysis and visualization scripts into various formats like static HTML pages, interactive graphs, summary pdf documents, and images. The on-card testing applet is compatible with cards starting from JC 2.1.1.

[1]The category “ICs, Smart Cards and Smart Card-Related Devices and Systems” contain around 35% (568 out of 1606) of all currently active certificates (Common Criteria, 2020).

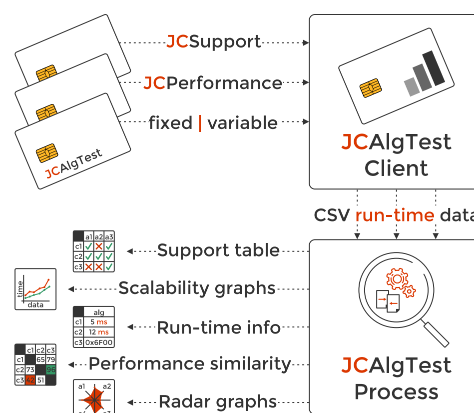

Figure 1: A high-level overview of the components and processes of our audit suite. The on-card applets measures the specification compliance and benchmarks the performance. Then, the client collects and processes the results to extract a useful information.

The development of the JCAlgTest suite started in 2007, in parallel with the collection of results from the cards available for testing. The initial implementation was done in C++ (host control application) and for Windows OS. After the code was ported to Java in 2012, we also started to obtain community-provided scans of cards, reaching more than 100 cards in the second half of 2021. The source code, releases, and the results collected are all available openly via GitHub and a dedicated webpage[2].

2.1 How to identify a smartcard?

The hardware used in smartcards is, to a great extent, defined by the ISO/IEC 7816 standard (ISO, 2013) and can have an outer form of a payment card, SIM card, or be embedded in a token. According to this standard, a typical smartcard must be tamper-resistant and should embed a general-purpose processor (8, 16, or 32-bits). The typical internal clock is between 20 to 50 MHz. Cards also features three types of memory: a Read-Only Memory (ROM) for the operating system, a Random Access Memory (RAM, typically below 10KB), and a larger application memory for binaries and data (20KB-1MB of EEPROM or FLASH). Typically, a true random number generator is available as a source of randomness. Finally, many cards also incorporate a *cryptographic co-processor* to provide hardware acceleration for computationally intensive cryptographic algorithms. The JavaCard platform established itself as the major programming interface, allowing to write applications (applets) in a restricted version of Java language portable between cards of different vendors.

[2]http://jcalgtest.org

If a user possesses a smartcard of (yet) unknown origin, there are multiple options on how to identify its manufacturer and type with a different level of robustness against the intentional manipulation of the metadata used to estimate the card's origin.

2.2 ATR and CPLC collection module

The main JCAlgTest host application lists all available readers and offers a connection to a specific smartcard. The Answer To Reset (ATR) bytes are automatically returned by card after the connection is established. Additionally, two APDU commands (*0x80CA9F7F* and *0x00CA9F7F* as defined by GlobalPlatform and ISO7816, respectively) are issued after the selection of CardManager entity to retrieve Card Production Life Cycle (CPLC) metadata. The retrieval is very fast and finishes in less than 3 seconds. The large majority of cards contain CPLC data, though some with obviously invalid/unfilled values like zeroes or invalid *ICFabricationDate* values.

2.3 Supported JavaCard algs module

The JavaCard API specification contains a multitude of packages with security-related classes of cryptographic algorithms (e.g., Cipher, Signature, MessageDigest, RandomData), further parameterized using different algorithm types (e.g., ALG_AES or ALG_RSA), key lengths, and modes of operation. In the JC API 3.0.5 (Oracle, 2015), the algorithm–parameter combinations add up to more than 200. However, a specific card product does not need to support all the algorithms specified in the documentation. Due to a large number of possible combinations, manual compliance testing is tedious and error-prone. Instead, JCAlgTest uses an on-card applet, trying to instantiate all defined classes and their corresponding parameters, and reports the results to the host application. Additionally, generic card information like the size of persistent and transient memory, reported JavaCard version, or maximum transaction commit capacity is also collected. The testing itself relies on the exceptions captured on-card, it is fully automated, and it completes in a relatively short time – usually within a few minutes. The results collected from all cards in the database are processed together to create a single large support matrix, listing all possible algorithms from specification and their support by a particular card. Selected aggregated results are provided in Table 1.

2.4 Performance collection module

Card performance varies greatly depending on the internal chip frequency, hardware-accelerated subset of algorithms, the timings of the memory chips used, and implementation choices (e.g., primality search algorithm). Despite that, smartcard manufacturers rarely provide extensive detailed reports on the performance of their products, or only incomplete ones (e.g., the raw speed of encryption engine without considering overhead for transferring data in and out and scheduling key for use).

In order to evaluate and compare cards for a specific use case, all individual algorithms are benchmarked for all possible parameter combination (e.g., various key lengths or padding options). Additionally, all methods of particular class (e.g., *setKey()*, *getKey()* and *eraseKey()* for class *AESKey*) are measured, resulting in more than 2300 possible combinations to be tested.

Unlike Java, JavaCard provides no direct on-card timer, and thus the time measurements must be performed by the host application on PC. As a result, we cannot measure only the targeted operation but are also measuring the time required to transfer data to and from the card and any other code executed before and after the operation.

To account for such a situation, we first run a *calibration phase*, which measures the overhead associated with all the non-target operations. During that phase, the target operation is not executed (the control flag is set to “false”), but all the surrounding code is. This enables us to measure the overhead of the functionality not related to the target function. Multiple calibration rounds are performed to account for variations in the measurements. Subsequently, the target operation is enabled (control flag set to “true”), and multiple experiment rounds are performed again. The target operation runtime is then obtained by simply subtracting the average overhead time (calibration phase) from the average runtime (experiment phase).

Even though this method provides accurate measurements for the great majority of the algorithms and methods, there are two cases that require special treatment:

- Methods that are preceded by other calls with inconsistent execution times (e.g., when RSA keypair generation is executed before signature operation).
- Operations with almost negligible overhead (<1ms) that is below the measurement error.

To address these specific cases, we increase the relative time of the target operation by wrapping it into the loop with a predefined number of iterations

(if necessary, the cryptographic key values used are alternated to prevent runtime optimizations by using the same value; compiled bytecode is also checked for unwanted compiler optimizations). All input buffers are pre-allocated to the maximum allowed length, and all objects to be used are pre-initialized to further reduce measurement variance. The resulting measurement precision is accurate enough to measure the performance of all the JavaCard methods, including utility methods with very low overhead (e.g., copying data between two RAM arrays). The whole set of performance experiments typically takes several hours to finish and depends not only on the card's speed but primarily on the number of supported algorithms, which increases the number of measurement sessions to be executed.

The performance testing runs in two modes, both providing the execution time in milliseconds for the specified length of the processed data. The first mode operates with fixed length – where sensible, we use 256 bytes data length, which corresponds to the length of the full APDU command. The second mode investigates the operation performed when applied to the data of different lengths – we test lengths of 16, 32, 64, 128, 256, and 512 bytes.

The performance testing methodology introduces non-trivial stress on a card and may result in a temporary freeze of the card (resolved by re-insertion of the card into a reader) and, in rare cases, also in permanent card blocking.

2.5 Supported JC packages module

Same as for cryptographic algorithms, no direct functionality is present to retrieve supported JavaCard packages. While support for a single algorithm can be tested by its instantiation and handling an eventual exception, support for the whole JavaCard package (e.g., *javacardx.crypto*) must be tested differently – testing applet will not even load to the card if it contains unsupported packages.

We instead utilize the behavior of an on-card verifier testing the bytes of the uploaded applet (*cap* file) during the LOAD command of GlobalPlatform specification and developed jcAIDScan tool[3]. A small, empty applet is first compiled and converted to *cap* file, ready to be uploaded to the card. Before the upload, cap file is unpacked (zip file structure) and internal file named *Import.cap* is (automatically) modified to reference additional package we like to test (package AID and its version, e.g., *0x020107A0000000620201* for *javacardx.crypto v1.2* as defined in JavaCard API

[3]https://github.com/petrs/jcAIDScan

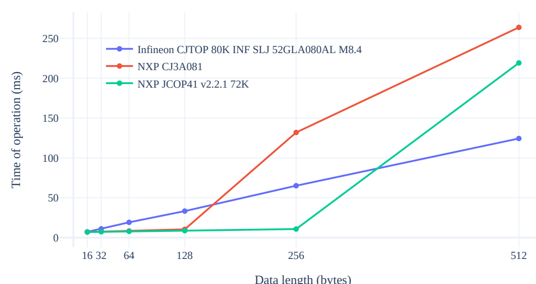

Figure 2: The example relationship between the input length with the runtime of 3DES with ISO9797 M1 padding in three different commercial JavaCards. The nonlinear increase for NXP CJ3A081 card between data with 128 and 256 bytes can be explained by memory limits of internal processing buffers, providing information about the hardware used.

2.2.1). The modified cap file with applet is attempted for upload – if no error is detected, the specific package and version is supported; otherwise, the support is missing. We test for all packages specified in JavaCard specification up to version 3.0.5 (*javacard.**, *javacardx.**, *org.globalplatform.**, *visa.openplatform.** and all their sub-packages), testing 120 different packages and versions in total. The testing can be easily extended to scan also for other packages if their AID is known. The whole process usually finishes below 10 minutes, depending on the card speed.

2.6 RSA/ECC keys collection module

While mostly random, the keys generated by a specific card may contain a small bias stemming from the implementation of the key generation algorithm (Svenda et al., 2016). The testing applet repeatedly calls *KeyPair.genKey()* method for instances of RSA and ECC asymmetric ciphers and export both public and private parts to the host. The extracted keys are later analyzed for the presence of yet unknown bias (bias discovery phase) or already known specific one (bias identification phase). The analysis is performed by other tools outside the keys collection module. In general, the discovery phase requires at least hundreds of thousands of keys and may take days or even weeks to complete, while the identification phase requires only 100-1000 keys collected within minutes or hours at most. The collection module supports parallel collection from multiple cards of the same type.

2.7 Presentation module

The raw data collected from a range of cards by the previously described modules are processed into a form more suitable for public display. CPLC meta-

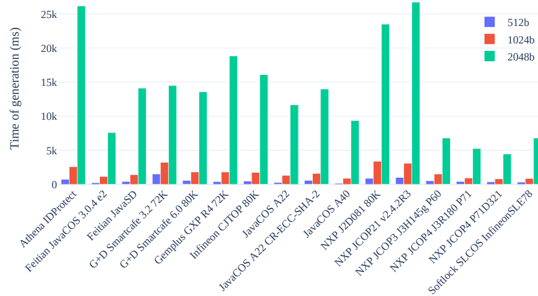

Figure 3: The average time required to generate a single RSA keypair of varying length on a selected subset of cards supporting all of the lengths. The observed run-time differences are due to the prime search algorithms executed in the cryptographic co-processor (Svenda et al., 2016)

data is parsed and converted into .dot format, later visualized by GraphViz software to highlight similar clusters. Supported algorithms are combined into a large matrix, with rows corresponding to the particular algorithm from JavaCard specification and columns corresponding to tested cards, all embedded in a static HTML page (Table 1 presents only aggregated results). Performance results are visualized in automatically generated HTML tables and graphs like Figure 2, showing speed dependency on the length of the data processed, or Figure 3, showing average time required to generate RSA keypairs of varying lengths.

3 Ecosystem insights

Based on a large number of results collected since 2007, we present an insight into the ecosystem as a whole. Approximately 50% of these cards were tested directly by us, while the rest were provided by volunteers worldwide. The longer and more detailed performance tests were performed only on cards available in our laboratory. Due to the nature of the study, we were able to include only smartcards that are commercially available in smaller quantities via third-party resellers. The on-premise testing was performed on an ordinary laptop and standard Gemalto IDBridge CT30 smartcard reader. The testing is designed to be independent of the host's performance.

3.1 JavaCard API support

The open database contains results for supported algorithms collected from more than 100 smartcards with different basic identification (e.g., product marketing title). Out of these, around ninety are different cards, with the rest being likely duplicates only enrolled under a different name. Since 1999 (JC API v2.1), the JavaCard API has defined DES and 3DES algorithms and TRNG as a source of random data. As seen in Table 1, almost all manufacturers provide support for these algorithms. On the other hand, while the AES algorithm (standardized by NIST in the year 2000) was specified already in version JC API v2.2.0 (2002), it was not until 2006 that some cards started to support it, while it became common only after the year 2010 (based on ICFabricationDate from the CPLC info). Almost all cards provide both ECB and CBC modes of operation for block ciphers, as well as for the padding schemes PKCS#1 and ISO9797. The advanced padding methods like OAEP and PSS are supported by less than 50% of the cards (but almost all recent ones with JC API v3.0.5).

JC API v2.1 also specifies RSA cryptosystem (up to 2048-bit keys), and although the majority of modern cards comply, very few products handle longer keys (e.g., 4096-bit ones). This is due to both the time requirements of longer keys generation and the universal shift towards elliptic curve cryptosystems that provide the same level of security with fewer key bits.

In the case of DSA, we observe only a small fraction of the cards with support. Newer cards seem to abandon DSA for its modern elliptic curve counterpart, ECDSA. Despite this, the overall picture is that EC algorithms are not yet fully supported in newer cards ($\geqslant$3.0.1), as, for instance, only around 70% of these cards expose EC key lengths of at least 256 bits. Moreover, only a hashed version of the Diffie-Hellman Key Agreement protocol (IEEE P1363) is frequently available, while implementations outputting both plain coordinates are only sparsely available (except for the newest JC API 3.0.5 cards). No card with mode for authenticated encryption (CCS, GCM) was encountered. Finally, manufacturers implement all classic hashing functions such as SHA-1, SHA-2 (256-bit), and RIPEMD-160, while support for the insecure MD5 seems to be phased out in the newer cards. The support for SHA-2 (512-bits) is common in newer cards, but no tested card yet supports SHA-3.

Additionally, the full-text search performed on all issued Common Criteria certificates (Common Criteria, 2017) for JavaCard-relevant keywords shows that the first card with JC API 2.1 was certified in 1999. A total of 29 certificates were issued for cards with JC API 2.2.2, with the first one certified in the year 2009. JC API 3.0.4 is now the dominant version with 19 active certificates. The first card with JC API 3.0.5 was certified in 2018, with a total of 4 certificates. No card with JC API 3.1 was certified to date. Performance of operations and support for higher JavaCard API is typically correlated, as can be seen in Figure 4.

Table 1: The level of support for algorithms specified in JavaCard API. For a given feature, the *version* column specifies the JavaCard specification that defined it first, while the subsequent columns show its availability in cards reporting particular supported version via the *JCSystem.getVersion*() method and maximally supported version of the *javacard.framework* package. Results for smartcards with an unknown version were not included.

Feature	**First in version**	**JC $\leqslant$ 2.2.1 (21 cards)**	**JC 2.2.2 (26 cards)**	**JC 3.0.1/2 (12 cards)**	**JC 3.0.4 (29 cards)**	**JC 3.0.5 (11 cards)**
Truly random number generator						
TRNG (ALG_SECURE_RANDOM)	$\leqslant$ 2.1	100%	100%	100%	100%	100%
Block ciphers used for encryption or MAC						
DES (ALG_DES_CBC_NOPAD)	$\leqslant$ 2.1	100%	100%	100%	100%	100%
AES (ALG_AES_BLOCK_128_CBC_NOPAD)	2.2.0	52%	96%	100%	100%	100%
KOREAN SEED (ALG_KOREAN_SEED_CBC_NOPAD)	2.2.2	5%	62%	75%	34%	0%
Public-key algorithms based on modular arithmetic						
1024-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_1024)	$\leqslant$ 2.1	76%	96%	100%	93%	82%
2048-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_2048)	$\leqslant$ 2.1	67%	96%	100%	93%	82%
4096-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_4096)	3.0.1	0%	0%	0%	3%	0%
1024-bit DSA (ALG_DSA LENGTH_DSA_1024)	$\leqslant$ 2.1	5%	8%	8%	10%	0%
Public-key algorithms based on elliptic curves						
192-bit ECC (ALG_EC_FP LENGTH_EC_FP_192)	2.2.1	5%	62%	83%	66%	82%
256-bit ECC (ALG_EC_FP LENGTH_EC_FP_256)	3.0.1	0%	50%	75%	66%	82%
384-bit ECC (ALG_EC_FP LENGTH_EC_FP_384)	3.0.1	0%	12%	17%	62%	82%
521-bit ECC (ALG_EC_FP LENGTH_EC_FP_521)	3.0.4	0%	4%	8%	45%	82%
ECDSA SHA-1 (ALG_ECDSA_SHA)	2.2.0	24%	84%	100%	69%	82%
ECDSA SHA-2 (ALG_ECDSA_SHA_256)	3.0.1	5%	12%	100%	69%	82%
ECDH IEEE P1363 (ALG_EC_SVDP_DH)	2.2.1	29%	81%	100%	69%	82%
IEEE P1363 plain coord. X (ALG_EC_SVDP_DH_PLAIN)	3.0.1	5%	4%	67%	48%	82%
IEEE P1363 plain c. X,Y (ALG_EC_SVDP_DH_PLAIN_XY)	3.0.5	0%	0%	0%	17%	82%
Modes of operation and padding modes						
ECB, CBC modes	$\leqslant$ 2.1	100%	100%	100%	100%	100%
CCM, GCM modes (CIPHER_AES_CCM, CIPHER_AES_GCM)	3.0.5	0%	0%	0%	0%	0%
PKCS1, NOPAD padding	$\leqslant$ 2.1	95%	100%	100%	100%	100%
PKCS1 OAEP scheme (ALG_RSA_PKCS1_OAEP)	$\leqslant$ 2.1	14%	31%	8%	41%	82%
PKCS1 PSS sheme (ALG_RSA_SHA_PKCS1_PSS)	3.0.1	14%	19%	83%	41%	100%
ISO14888 padding (ALG_RSA_ISO14888)	$\leqslant$ 2.1	14%	12%	8%	0%	0%
ISO9796 padding (ALG_RSA_SHA_ISO9796)	$\leqslant$ 2.1	81%	100%	100%	86%	100%
ISO9797 padding (ALG_DES_MAC8_ISO9797_M1/M2)	$\leqslant$ 2.1	90%	100%	100%	100%	100%
Hash functions						
MD5 (ALG_MD5)	$\leqslant$ 2.1	90%	77%	92%	62%	0%
SHA-1 (ALG_SHA)	$\leqslant$ 2.1	95%	100%	100%	100%	100%
SHA-256 (ALG_SHA_256)	2.2.2	14%	88%	100%	97%	100%
SHA-512 (ALG_SHA_512)	2.2.2	5%	23%	25%	90%	100%
SHA-3 (ALG_SHA3_256)	3.0.5	0%	0%	0%	0%	0%

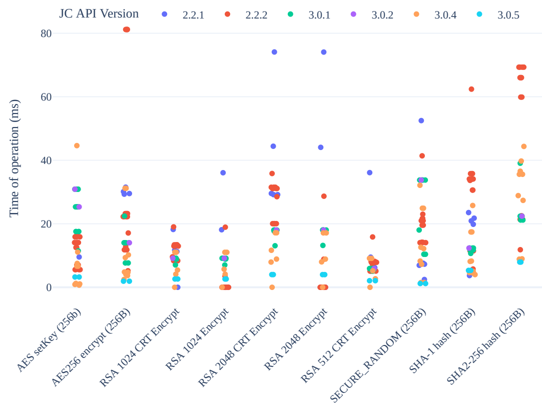

Figure 4: Performance clusters of the most common cryptographic operations that take less than 100 milliseconds to execute. Every point corresponds to a single operation executed on a particular card with a particular version of JavaCard API. Only an example subset of all tested cards is shown for clarity.

3.2 JavaCard packages support

In total, 40 cards were analyzed for the available packages from JavaCard API up to JC API 3.0.5 using jcAIDScan module. While standard packages like *javacard.framework* or *javacardx.crypto* are always supported (in version corresponding to the card's API compliance), more specialized packages like *javacardx.biometry* (11x), *javacardx.framework.util.intx* (9x) *javacardx.framework.math* (4x) are supported only in relatively small fraction of cards.

The jcAIDScan tool also test packages from *visa.openplatform* and its successor *org.globalplatform*. However, we have not yet collected enough results to provide representative statistics.

3.3 Card Production Life Cycle clusters

A total of 71 cards were recorded together with its Card Production Life Cycle (CPLC) information. We analyzed the available security certificates and other public sources to obtain the mapping between the numeric constants used and their human-readable titles of manufacturers (ICFabricator), chip type (ICType), and operating system (OperatingSystemID). In total, we encountered:

- **ICFabricator:** Nine different chip fabricators: Renesas (0003, 3060), Infineon (0005, 4090, 4830), Philips (2050), NXP (4070, 4790), Atmel (4180), Samsung (4250), STMicro (5354), Tongxin (008c) and one unidentified (4220).
- **ICType:** 45 different chip types with NXP and Infineon responsible for the large majority of the chips encountered. While NXP tends to manufacture hardware chips together with the whole software stack and market resulting cards under its own brand, Infineon as a chip manufacturer provides not only complete cards sold under its brand but more frequently provides chips to other vendors.
- **OperatingSystemID:** 23 different codes for operating systems with 52 sub-variants.

We also analyzed the current market situation with respect to existing vendors after various company mergers and visualized the resulting aggregate accordingly. For example, the *Gemalto* cluster is aggregating not only cards produced under the *Gemalto* brand, but also for *Gemplus*, *Axalto* and *Schlumberger* (now inactive) brands. The full visualization and additional artifacts are available[4].

4 Related work

Many papers are investigating the JavaCard platform, with the great majority of them examining its security from either the software perspective (Besson et al., 2014; Farhadi and Lanet, 2017; Lancia and Bouffard, 2016; Laugier and Razafindralambo, 2015; Volokitin and Poll, 2016) or the hardware one (Barbu et al., 2010; Kasmi et al., 2015; Kocher et al., 1999; Vermoen et al., 2007). A smaller number of works focus on performance and capabilities. The performance of basic cryptographic primitives on three programmable smartcard platforms (JavaCard, .NET, and MultOS) is examined in (Hajny et al., 2014), but focused only small subset of operations required for computation of group signatures. (Bernabé and Clarke, 2013) study RSA performance in JavaCards with a focus on predicting the operation run-time of RSA-4096 functions. The MESURE project (Bouzefrane et al., 2008), founded by the French government and started in 2006, produced a complex benchmarking framework. Exact timing measurements were not published, only an aggregate score for each card, likely due to nuances of the performance comparison. Faster computation may also result in less secure implementations (Nemec et al., 2017; Jancar et al., 2020). The open-source GlobalPlatformPro tool by M. Paljak (Martin Paljak, 2020) provides human-readable output for CPLC data retrieved from a connected smartcard. The open database of community collected mappings between ATR and card name is maintained (Ludovic Rousseau, 2009).

[4] https://crocs.fi.muni.cz/papers/jcalgtest_secrypt22

5 Conclusions

We proposed an improved certification procedure with additional artifacts collected by evaluation facilities during the certification process using open-source tools, later verifiable by end-users without need for specialized knowledge or privileged access to smartcard hardware specification or software source code (black-box testing). To support this vision, we developed a suite of open tools for the collection of metadata, functionality, and performance parameters, and specific properties of generated cryptographic keys by the tested cards.

We also continuously maintained the largest open database with the respective results using a combination of cards from our laboratory and provided by community effort, totaling more than 100 cards. Such a database not only provides insight into the ecosystem of cryptographic smartcards spanning over almost two decades but also makes the results for common smartcards accessible – providing data for end-user verification may be included in the certification process.

Acknowledgement: P. Svenda and A. Dufka were supported by Ai-SecTools (VJ02010010) project. Earlier support for JCAlgTest project was provided by the European cybersecurity pilot CyberSec4Europe.

REFERENCES

Barbu, G., Thiebeauld, H., and Guerin, V. (2010). Attacks on java card 3.0 combining fault and logical attacks. *Smart Card Research and Advanced Application*, pages 148–163.

Bernabé, G. and Clarke, N. (2013). Study of RSA performance in Java Cards. *Advances in Communications, Computing, Networks and Security Volume 10*, page 45.

Besson, F., Jensen, T., and Vittet, P. (2014). Sawjacard: a static analysis tool for certifying Java Card applications. In *International Static Analysis Symposium*, pages 51–67. Springer.

Bouzefrane, S., Cordry, J., and Paradinas, P. (2008). A methodology for testing Java Card performance. In *CFSE'08 Conférence Française en Systèmes d'Exploitation, Suisse*.

Farhadi, M. and Lanet, J.-L. (2017). Chronicle of a Java Card death. *Journal of Computer Virology and Hacking Techniques*, 13(2):109–123.

Hajny, J., Malina, L., Martinasek, Z., and Tethal, O. (2014). Performance evaluation of primitives for privacy-enhancing cryptography on current smartcards and smart-phones. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 17–33. Springer.

ISO (2013). ISO/IEC 7816-4:2013 - identification cards – integrated circuit cards – part 4: Organization, security and commands for interchange. International Organization for Standardization.

Jancar, J., Sedlacek, V., Svenda, P., and Sys, M. (2020). Minerva: The curse of ECDSA nonces. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):281–308.

Kasmi, M. A., Azizi, M., and Lanet, J.-L. (2015). Side channel analysis techniques towards a methodology for reverse engineering of Java Card byte-code. In *Information Assurance and Security (IAS), 2015 11th International Conference on*, pages 104–110. IEEE.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer.

Lancia, J. and Bouffard, G. (2016). Fuzzing and overflows in Java Card smart cards. In *SSTIC Conference, Rennes, France*.

Laugier, B. and Razafindralambo, T. (2015). Misuse of frame creation to exploit stack underflow attacks on java card. In *International Conference on Smart Card Research and Advanced Applications*, pages 89–104. Springer.

Common Criteria (2017). Common Criteria for information technology security evaluation, version 3.1, revision 5, ccmb-2017-04-001. https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf.

Common Criteria (2020). Certified products list - statistics (retrieved 2022-04-11). https://www.commoncriteriaportal.org/products/stats/.

Ludovic Rousseau (2009). Smart card ATR parsing (retrieved 2022-04-11). https://smartcard-atr.apdu.fr/.

Martin Paljak (2020). GlobalPlatformPro v20.01.23. https://github.com/martinpaljak/GlobalPlatformPro.

Oracle (2015). Java Card 3.0.5 platform specification. https://docs.oracle.com/javacard/3.0.5/index.html.

Nemec, M., Sys, M., Svenda, P., Klinec, D., and Matyas, V. (2017). The return of Coppersmith's attack: Practical factorization of widely used RSA moduli. In *ACM CCS 2017*, CCS '17, pages 1631–1648, New York, NY, USA. ACM.

Svenda, P., Nemec, M., Sekan, P., Kvasnovsky, R., Formanek, D., Komarek, D., and Matyas, V. (2016). The million-key question – investigating the origins of RSA public keys. In *The 25th USENIX Security Symposium (UsenixSec'2016)*, pages 893–910. USENIX.

NIST (2014). Validated FIPS 140-1 and FIPS 140-2 cryptographic modules. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm.

Vermoen, D., Witteman, M., and Gaydadjiev, G. (2007). Reverse engineering java card applets using power analysis. *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 138–149.

Volokitin, S. and Poll, E. (2016). Logical attacks on secured containers of the java card platform. In *International Conference on Smart Card Research and Advanced Applications*, pages 122–136. Springer.