The adoption rate of JavaCard features by certified products and open-source projects

Lukas Zaoral[1], Antonin Dufka[2], and Petr Svenda[2]

[1] Red Hat
lzaoral@redhat.com
[2] Masaryk University
dufkan@mail.muni.cz, xsvenda@fi.muni.cz

Abstract. JavaCard is the most prevalent platform for cryptographic smartcards nowadays. Despite having more than 20 billion smartcards shipped with the JavaCard virtual machine and thirteen revisions since the JavaCard API specification was first published more than two decades ago, uptake of newly added features, cryptographic algorithms or their parameterizations, and systematic analysis of overall activity is missing. We aim to fill this gap by mapping the activity of the JavaCard ecosystem from publicly available sources with a focus on 1) information available from security certification documents available under Common Criteria and FIPS140 schemes and 2) activity and resources required by JavaCard applets released in an open-source domain[3].
The analysis performed on all certificates issued between the years 1997-2023 and on more than 200 public JavaCard applets shows that new features from JavaCard specification are adopted slowly. It typically takes six or more years before a majority of certified products add corresponding support. Open-source applets utilize new features even later, likely due to the unavailability of recent performant smartcards in smaller quantities. Additionally, almost 70% of constants defined in JavaCard API specification are completely unused in open-source applets. The applet portability improves with recent cards, and transient memory requirements (scarce resource on smartcards) are typically small. While around twenty products have been consistently certified every year since 2009, the open-source ecosystem became more active around 2013 but seemed to decline in the past two years. As a result, the whole smartcard ecosystem is likely negatively impacted by limited exposure to new ideas and usage scenarios, serving only well-established domains and potentially harming its long-term competitiveness with other technologies.

Keywords: Smartcard · JavaCard · Security certification · Open-source

1 Introduction

A recent paper [12] mapped the difference between features (cryptographic algorithms, key lengths) described in JavaCard API specification and features

[3] Paper supplementary materials, full results of analysis and open tools are available at https://crocs.fi.muni.cz/papers/cardis2023.

actually supported by the real-world physical smartcards and documented a large disparity. As implementation of cryptographic algorithms frequently requires dedicated hardware co-processors to achieve acceptable performance and secure execution, the decision about the level of support is left to a smartcard vendor, with API specification intentionally listing the majority of the features as *optional* to allow for such flexibility. But while JavaCard applets are supposed to be binary portable between different smartcards, the uneven level of feature support directly influences the actual portability and adoption of specific algorithms by applets developed for the smartcards.

The analysis performed in paper [12] has two main limitations – although more than 100 smartcards are analyzed, only the ones available to community-maintained JCAlgTest database are considered (typically missing the recent or uncommon smartcards), and actual use of algorithms in applet code running on these cards is not evaluated. As a result, the impact of uneven support of JavaCard API features on the whole JavaCard ecosystem and the rate of new features adoption are not studied enough. We aim to fill this gap by answering the following research questions:

Q1: *What is the level and delay in adoption for cryptographic algorithms introduced by specific versions of JavaCard specification by a) certified smartcards and b) implementation of open-source applets?*
Q2: *How does the JavaCard open-source ecosystem evolve with respect to development activity, requested JavaCard features, and memory utilization?*

To answer these questions, we first performed an automatic analysis of documents accompanying security certification performed under the Common Criteria and FIPS140 schemes for JavaCard API-related keywords and analyzed the evolution in time for different API versions and cryptographic algorithms included. That still left one angle unanswered – what algorithms are typically required by applications (applets) running on these smartcards?

We addressed this question by analysis of more than 200 open-source JavaCard projects. The resulting insight covers the evolution of cryptographic algorithms in time, the expected time span before the algorithms from the given version of JavaCard specification start to be available in practice, and the overall activity of the JavaCard open-source community. The results can be used to inform the creators of the JavaCard specification regarding the features popularity and the possibility of features deprecation, the vendors of smartcards with respect to the support for new features and comparison with competition, and the developers with respect to the expected time of practical availability of features included in a new version of the specification.

Paper contribution:

- Analysis of JavaCard API features adoption rate by products certified under Common Criteria and FIPS140 schemes.
- Analysis of the evolution of the JavaCard open-source ecosystem of more than 200 projects with respect to resources required (packages, crypto. algs., memory) and applets portability between multiple physical smartcards.

The rest of this section introduces the JavaCard specification and surveys the related work. Section 2 analyzes mentions of JavaCard technology by certification artifacts issued under Common Criteria and FIPS140 certification schemes. Section 3 surveys the JavaCard open-source ecosystem and analyzes its activity in time and the portability of applets between different physical cards. Section 4 analyzes their memory and cryptographic resource requirements. Discussion about observed trends and limitations is provided in Section 5 with conclusions following in Section 6.

1.1 Related work

The closest work to ours is a recent publication by Svenda et al. [12]. In contrast to this paper, only support of JavaCard smartcards by direct testing is performed in [12], certification reports are not analyzed and usage of the JavaCard features in actual source code is not analyzed. A work by Hajny et al. [6] presented performance measurement of selected basic cryptographic operations on the three major smartcard platforms (JavaCard, .NET, and MultOS).

Apart from works focusing on smartcards, there have been a number of works analyzing various software ecosystems. A systematic review of software ecosystem literature was given in works by Barbosa and Alves [1] and Manikas and Hansen [7]. An analysis focusing on mobile software ecosystems was performed by Fontão et al. [3]. Ecosystem of open-source software for drones was surveyed by Glossner et al. [5]. Furthermore, there have been a number of analyses performed by mainstream developer platforms like StackOverflow[4] and GitHub[5] who produce yearly reports of their ecosystems and developers.

2 JavaCard in Common Criteria and FIPS140

The necessary prerequisite for an algorithm to be used in a production applet's code is its support by the used smartcard platform. As smartcards tend to be used in environments imposing strict compliance requirements, the platform often needs to be certified, typically under FIPS140 or Common Criteria schemes.

We therefore performed an analysis of certification documents for any mentions of JavaCard API versions and algorithms to obtain further insight into the current state of the certified devices ecosystem. These results support open-source applet analysis presented later in Section 4. While existing work [12] analyzed supported algorithms by direct testing of physical cards, only cards available to authors were tested. The analysis based on certification documents is therefore complementary to [12], as we analyze not only certificates for the JavaCard platform and cryptographic libraries but also certificates for applets built atop a base platform. Only certified items are analyzed here while the authors of [12] test also smartcards with non-certified platforms (although frequently based on certified underlying hardware).

[4] `https://survey.stackoverflow.co/2023/`.
[5] `https://octoverse.github.com/`.

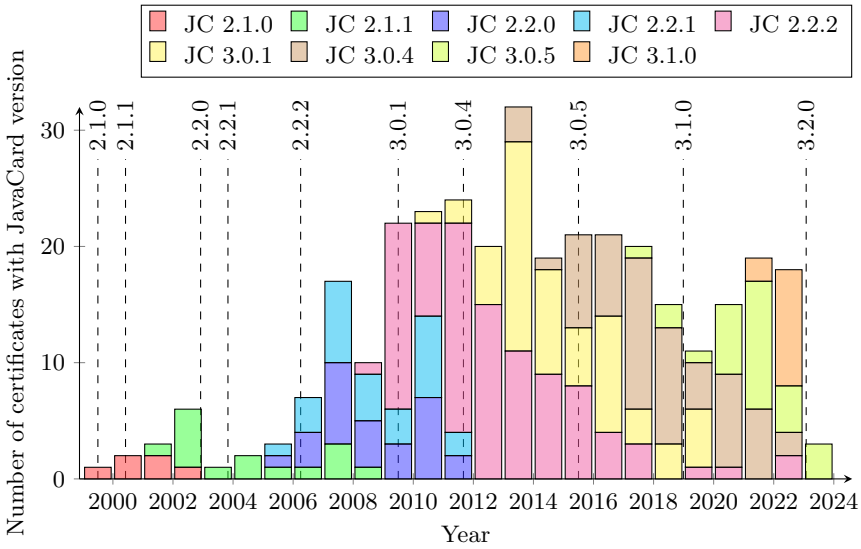

Fig. 1. The number of certification documents mentioning specific JavaCard API version per year (the year 2023 only till June). In case multiple versions were detected in a document, only the latest one was included in the plot.

To perform the analysis, we downloaded all public certification documents, conveniently collected by the Sec-certs project[6]. The dataset already contained JavaCard API version numbers extracted from the documents using regular expressions. However, the second part of the analysis required more precise data processing, which involved semi-manual filtering of raw data. We matched the regular expression `"ALG_[0-9A-Z_]+"` against text content of all certification documents and filtered only those that matched the pattern and corresponded a JavaCard target (list of all matched certificates can be found in Table 5 in the Appendix). In case a match was found, we compared the matching string to known JavaCard API names and corrected values that were clearly incorrectly read or input.

2.1 Analysis of certification documents

First, we analyzed the dependency of mentions of JavaCard API versions in certification documents on a certificate issuance year. The distribution is shown in Figure 1, where we display only the latest version detected per certificate document (338 documents in total). We see that JavaCard API version 2.2.2 was in use for the longest period, starting in 2008, 2 years after its introduction, and being commonly mentioned till 2017. Even in the year 2022, certificates mentioning this API version were issued. Still, since the year 2010, API with

[6] `https://seccerts.org/`.

major version number 3 started gradually appearing, and by 2013, they were mentioned in the majority of JavaCard certification documents. For the analysis of new algorithm adoption, we focused on these newer versions, namely 3.0.1, 3.0.4, 3.0.5, and 3.1.0.

Among the 9872 certificate documents in the dataset, we have identified 59 that describe a JavaCard target and include references to strings prefixed with `ALG_`, typical naming of JavaCard algorithm constants. Out of the 59 documents, those issued before the year 2009 were certified only under FIPS140 scheme (6 certificates). After that, all subsequent documents (53 in total) were issued under the Common Criteria scheme. In the documents, we identified 136 unique strings that match the pattern; however, only 103 of them correspond to algorithms included in the JavaCard API[7].

Figure 2 displays mentions of JavaCard algorithms introduced in JavaCard API 3.0.1 and later grouped by the certificate issuance year. In general, algorithms start being mentioned in certification documents only two or more years after the specification version is published. The only exception to this is the certificate of an Oberthur smartcard[8], which mentions algorithm `ALG_SHA_224` even before its specification was released.

JavaCard API version 3.0.5 was released in 2015, and the first algorithm added to the specification appeared in a certification document in 2017. Later, an algorithm (`ALG_EC_SVDP_DH_PLAIN_XY`), allowing for efficient ECDH computation outputting full point, started appearing in 2018. This algorithm was the most frequently mentioned in the documents in the year 2022, together with its variant (`ALG_EC_SVDP_DH_PLAIN`) outputting only X-coordinate that was introduced in API version 3.0.1. Version 3.0.4 introduced two new algorithms, which first appeared in certification documents in 2016, five years after its publication. Apart from the Oberthur anomaly, mentions of algorithms introduced in JavaCard API version 3.0.1, released in May 2009, started appearing in 2011, but their mentions became more frequent only after 2015.

We also inspected the remaining strings that matched the JavaCard algorithm pattern but are not part of the JavaCard API. Many of these algorithms are likely incorrectly input in the source documents; however, some of them clearly denote algorithms unsupported by JavaCard API. The algorithm names are presented in Table 1 with several interesting items like `ALG_EC_SVDP_DH_GK` whose function is not known due to missing public documentation, referring to Oberthur's proprietary API extensions[9]. Another interesting mentions are `ALG_ED25519PH_SHA_512` in four certificates by NXP[10], which most likely refers to Ed25519 signature algorithm with prehashed input and SHA512 function. The same four certificates also include string `ALG_MONT_DH_25519`, likely referring to Diffie-Hellman key agreement on Curve25519 in the Montgomery form.

[7] JavaCard API contains in total 151 constants with such name.

[8] `https://seccerts.org/fips/6d094db49a6e2242/`.

[9] `https://seccerts.org/fips/6d094db49a6e2242/`, `/a5bef651c8e3fd6c/`.

[10] `https://seccerts.org/cc/03aded94fb04c62e/`, `/45098872448f5816/`, `/03aded94fb04c62e/`, `/b0e6f667d52402df/`.

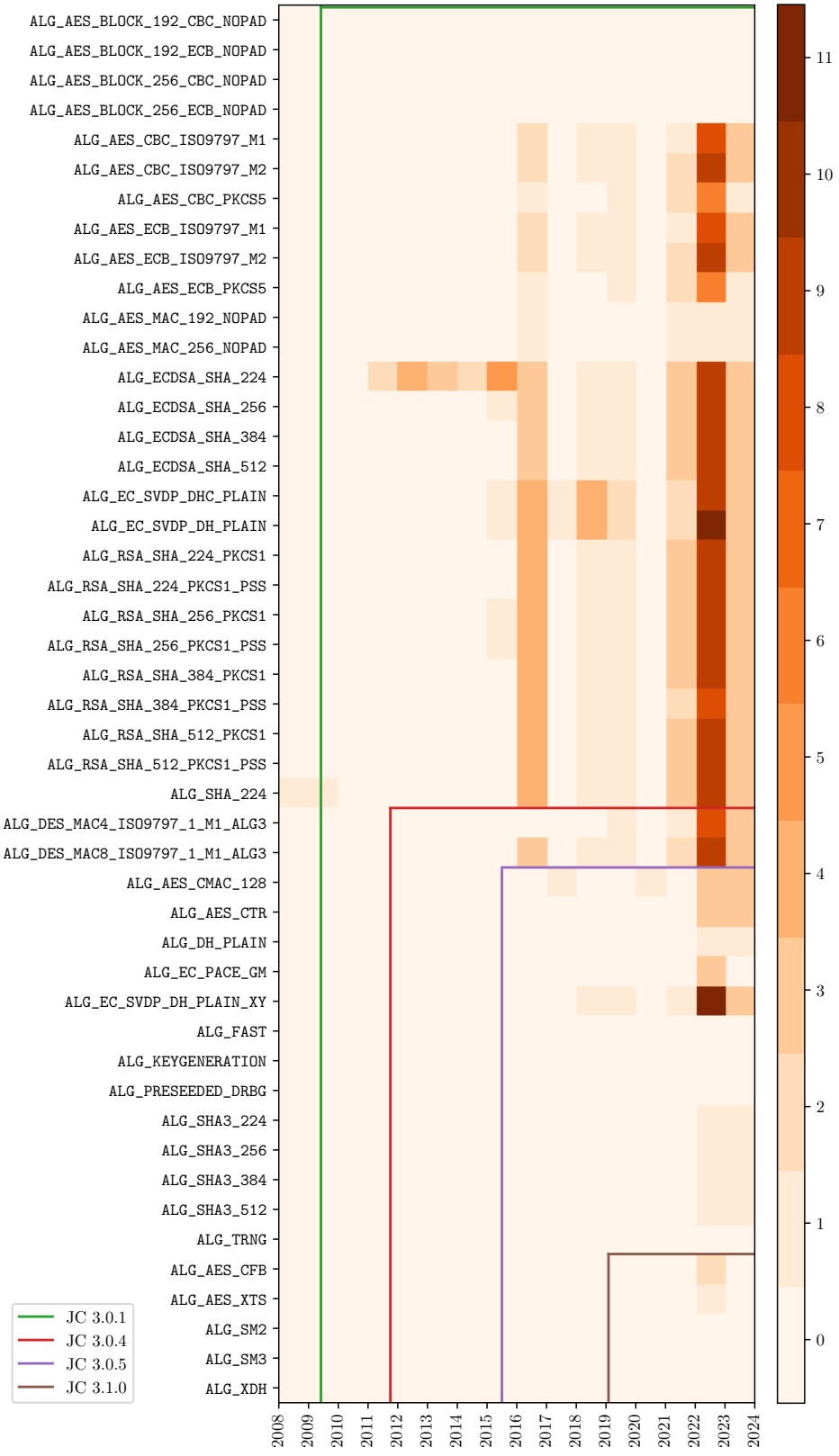

Fig. 2. Number of the Common Criteria and FIPS140 certificates that mention JavaCard API 3.0.1+ algorithms, by year.

Table 1. Algorithms detected in CC and FIPS certificates by search of prefix `ALG_` that are not included in the official JavaCard API specification.

ALG_AES_BLOCK_128_CBC_NOPAD_STANDARD	ALG_DES_CMAC8	ALG_EC_SVDP_DHC_GK
ALG_AES_CBC_ISO9797_M2_STANDARD	ALG_DES_ECB_PKCS7	ALG_EC_SVDP_DHC_PACE
ALG_AES_CBC_ISO9797_STANDARD	ALG_DES_MAC128_ISO9797_1_M2_ALG3	ALG_EC_SVDP_DH_GK
ALG_AES_CBC_PKCS7	ALG_DES_MAC_8_NOPAD	ALG_ED25519PH_SHA_512
ALG_AES_CMAC128	ALG_ECDSA_RAW	ALG_MONT_DH_25519
ALG_AES_CMAC16	ALG_ECDSA_SHA224	ALG_RSA_SHA256_PKCS1
ALG_AES_CMAC16_STANDARD	ALG_ECDSA_SHA256	ALG_RSA_SHA256_PKCS1_PSS
ALG_AES_CMAC8	ALG_ECDSA_SHA256_LDS	ALG_RSA_SHA_1_RFC2409
ALG_AES_ECB_PKCS7	ALG_ECDSA_SHA384	ALG_RSA_SHA_256_ISO9796
ALG_AES_MAC_128_ISO9797_1_M2_ALG3	ALG_ECDSA_SHA384_LDS	ALG_SHA2_CHAIN
ALG_DES_CBC_PKCS7	ALG_ECDSA_SHA_LDS	ALG_SHA_CHAIN

3 JavaCard open-source ecosystem

To analyze the overall focus, activity, and evolution of the JavaCard open-source ecosystem, we built a database of all available public repositories relevant to the JavaCard platform, performed a static and dynamic analysis of contained JavaCard applets, and tested deployability on several physical smartcards.

3.1 Database of open-source projects

Since 2017, we periodically searched for JavaCard open-source repositories on GitHub, SourceForge, and GitLab for the occurrence of *'javacard.framework'* keyword and collected hits into a public curated list hosted on GitHub[11]. We include almost all discovered repositories without assessing their maturity – only obviously trivial, testing or unfinished applets are excluded. As some projects are occasionally moved or removed, we create backups by forking each repository when included. While a large majority ($> 95\%$) of records were inserted by us, some contributions were also made by the community.

We have manually examined all repositories from the curated list to create a refined list with an explicit enumeration of JavaCard applets that we have found. The refined list contains 139 repositories containing at least one JavaCard applet or library. The repositories are mainly found on GitHub, with a very small number on SourceForge as well, and no repository has been detected on GitLab so far. Many repositories are monorepos with multiple projects. Sources that could not be parsed even with import resolution disabled were omitted from further analysis. The list contains 206 JavaCard projects with 223 possible applet entry points, of which 36 cannot be parsed precisely due to missing dependencies or programming mistakes preventing code compilation with standard *javac* compiler. For analysis of the usage of specification features, we excluded the applet corresponding to JCAlgTest project[12], which is designed to test the target smartcard and intentionally references (almost) all cryptographic algorithms and parameterization constants found in the JavaCard specification [9].

[11] `https://github.com/crocs-muni/javacard-curated-list`.
[12] `https://github.com/crocs-muni/jcalgtest`.

3.2 Activity of JavaCard open-source ecosystem in time

To analyze the activity of the JavaCard open-source ecosystem in time, we analyzed each project's state with respect to its git commits closest to (but prior) the next date interval. For a yearly activity, we analyzed the commit closest to the 1st of January of the next year, whereas for monthly activity, we used the commit closest to the 1st day of the next month.

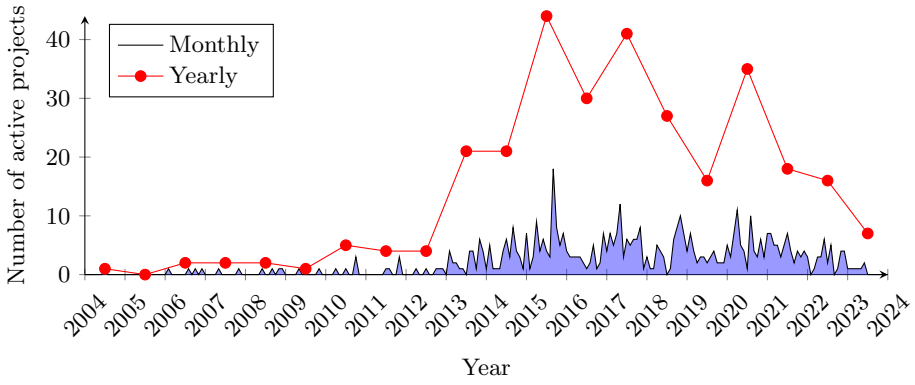

Fig. 3. Number of open-source projects with at least one commit per month (black line) or per year (red line) respectively. The year 2023 is only till end of June.

The number of projects with at least one commit in a given period (year or month) is shown in Figure 3. While the oldest commits are from the year 2004, the number of active projects was very low (around three projects) until the year 2013, when the number of active projects increased significantly to more than 20 projects, reaching more than 40 active projects in years 2015 and 2017. Figure 4 visualizes included repositories based on a number of forks and stars with the most popular repositories annotated.

While the development activity fluctuates significantly between years (the year 2019 had only around 50% of active projects with respect to 2017 and 2020), there seems to be a significant activity decrease following the year 2020, with only 7 active projects till the first half of the year 2023. Would the trend continue, we may be witnessing a decline of open-source activity in the JavaCard ecosystem, despite a still high number of newly certified JavaCard products (see Section 2).

An intriguing possibility might be a correlation with the trends observed in the JavaCard-related certification artifacts but with a lag of about four to six years. The lag can be explained by the delayed availability of capable certified smartcards to open-source developers for purchase in smaller quantities. If this correlation is genuine, we may observe increased activity in the open-source ecosystem in the following years again, as activity in the certification ecosystem dipped in the year 2019 and increased again since then.

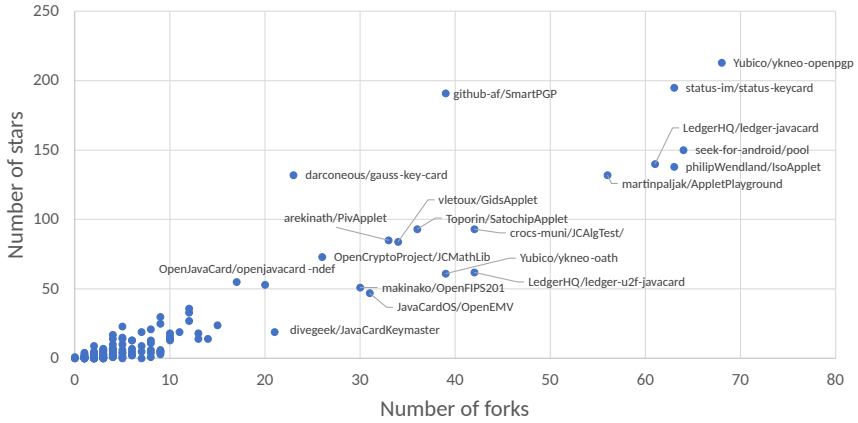

Fig. 4. The scatter plot of JavaCard repositories popularity using number of forks and stars on the GitHub platform. Projects with at least 20 forks are annotated by name.

3.3 Open-source applets compatibility and portability

To test the possibility of deployment of open-source applets to physical smartcards, we attempted the following steps for all open-source applets from the open-source database described in Section 3.1:

1) **Compile** applet using standard *javac* compiler, resulting in **.class* file(s). The step fails if the source code is not compatible with Java language (typically 1.8) or references proprietary packages.
2) **Convert** compiled **.class* files using Oracle JavaCard *converter* for specific version of JC SDK, resulting in a **.cap* file. The step fails if the bytecode violates JavaCard language specification or references proprietary packages.
3) **Upload** contents of the *cap* file to smartcard using the GlobalPlatform interface via *gppro* tool [10]. The step fails if *cap* file references unsupported packages or is rejected by the on-card verifier for other reasons.
4) **Install** applet instance from previously uploaded package (content of the *cap* file). The step fails if an exception is emitted in the applet's constructor, typically caused by instantiation of an unsupported algorithm (e.g., `ALG_SHA3_512`), unsupported parameters (e.g., of `LENGTH_RSA_4096` or invalid curve domain parameters), or excessive memory allocation (typically RAM memory via `JCSystem.getTransientByteArray()`). Occasionally, the exception may be caused by other programming mistakes or platform limitations (e.g., the limited size of the card's transaction buffer).
5) **Select** the installed applet instance, making it active for subsequent commands from a host controller. The step typically fails when additional custom code like delayed allocation or reset of the temporary state is executed.

The next step is performed only if the previous one succeeds. Steps 1) and 2) do not require any physical smartcard. Steps 3), 4), and 5) were performed on three physical smartcards and on the virtual card via jCardSim [2].

3.4 Applet compatibility with a simulated card

We first analyzed the possibility of performing applet conversion resulting in *cap* file, but with load, install, and select steps performed only for the jCardSim simulator. Such test detects applets that are compatible with (some) version of JavaCard specification and can be converted into a valid *cap* file. Yet no limitations of actual support of the required algorithms by the target physical card are imposed. Additionally, applets with programming errors that could be fixed easily (e.g., missing typecast from int to short, etc.) are listed in a separate column. The results are shown in Table 2.

Table 2. The number of applets with successful finalization of particular development step on virtual card simulated by jCardSim. Steps 1) and 2) are independent of any smartcard; steps 4) and 5) are tested on a simulated card provided by jCardSim.

Step	Count	(with fixes)
1) Compilation	174	187
2) Conversion (any JC SDK version)	145	176
3) Upload (does not apply for jCardSim)	-	-
4) Installation	121	150
5) Applet selection	111	144

Table 3. The portability of applets to different smartcards. The jCardSim simulator is listed as a theoretical upper bound of convertable and executable applets without resource limitations imposed by the particular physical platform. The numbers presented in this table also correspond to the applets with success/failure of measurement of entry point class constructor memory usage on physical smartcards. All skips correspond to applets that require a newer JavaCard API than supported by a given card.

Card	API	Success	Failure			Skip
			Upload	Install	Select	
jCardSim simulator	3.0.5	144	0	0	0	0
NXP JCOP4 J3R180 (2020)	3.0.5	124	7	13	0	0
Feitian JavaCOS A22	3.0.4	98	3	41	0	2
G+D StarSign Crypto USB Token S	3.0.4	64	3	73	1	2

3.5 Applet portability between physical cards

Secondly, we tested the *deployability* and *portability* of applets among three different smartcards, all relatively commonly available to open-source developers. The results are shown in Table 3 with the number of applets deployable to cards simulated via jCardSim serving as a baseline. The largest number of applets (124) are deployable to NXP JCOP4 J3R180 (CC certificate[13] issued

[13] `https://seccerts.org/cc/2a45531c2dbd1ab8/`.

on 01. 03. 2020), which is currently the most performant card available at online shops (2023 YTD) in small quantities. The decreasing number of deployable applets to other smartcards illustrates the limited portability of open-source applets among smartcards of different vendors.

4 Resources required by JavaCard applets

Having the large list of open-source JavaCard projects available via git repositories allows us to analyze the packages, classes, methods, and constants using static analysis of their source code. The analysis relies heavily on the functionality provided by the Spoon library [11] designed for analysis and transformation of Java source code. Spoon parses the input source code into a complete abstract syntax tree (AST). Therefore, the source code can be analyzed and instrumented more reliably than in the case of the regex-based approaches searching for target keywords. The results are provided in Section 4.1

The Spoon library also allows for automatic instrumentation of the applet's code usable for the dynamic analysis of memory requirements by the target applet with results presented in Section 4.2.

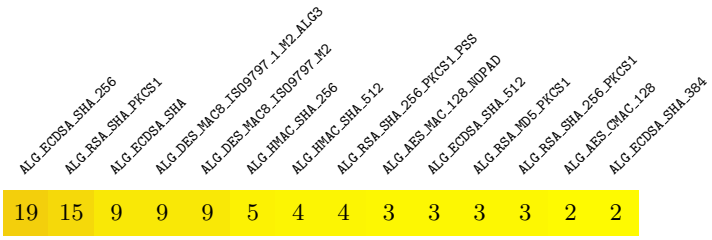

Fig. 5. `javacard.security.Signature` `ALG` constants with at least two usages.

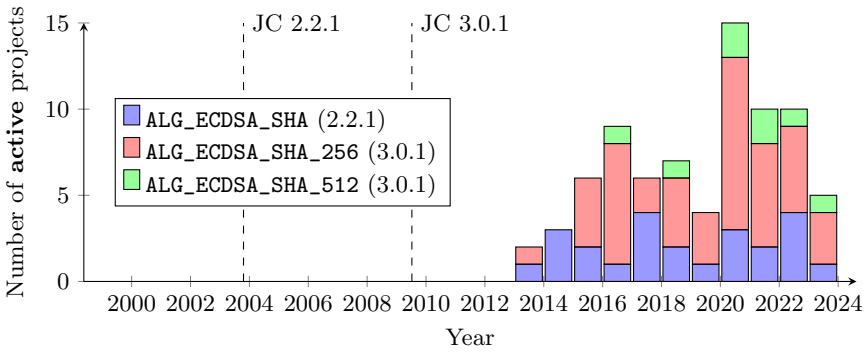

Fig. 6. Histogram of the adoption of `ALG_ECDSA_SHA` constants.

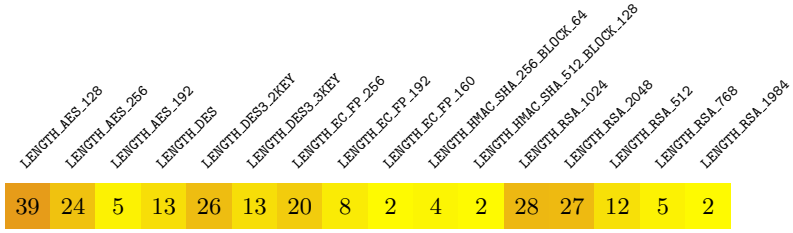

Fig. 7. `javacard.security.KeyBuilder` `LENGTH` constants with at least two usages sorted by the corresponding algorithms.

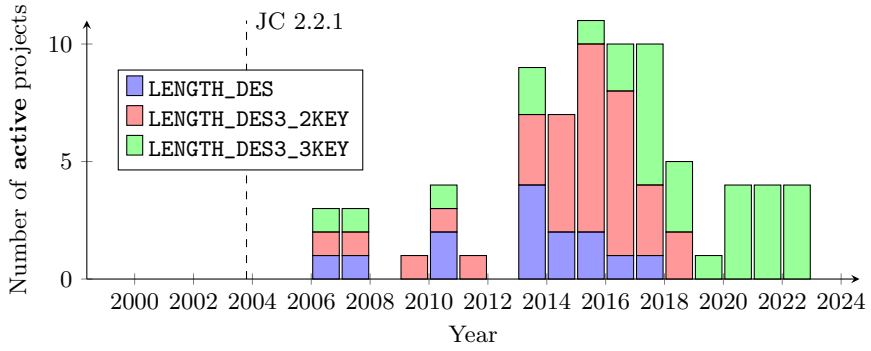

Fig. 8. Histogram of the adoption of `LENGTH_DES` constants from JavaCard 2.2.1.

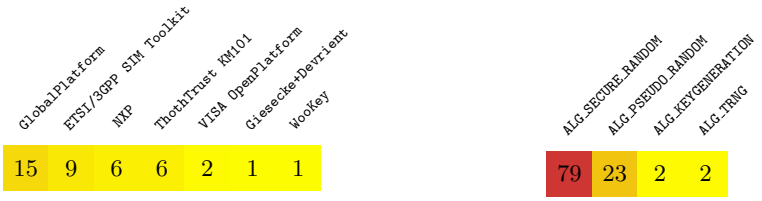

Fig. 9. Usage of third-party APIs and `javacard.security.RandomData` constants.

4.1 JavaCard packages and cryptographic algorithms required

The combination of git history and algorithms extracted from the source code at the given date (e.g., the latest commit in a given year) allows us to analyze the rate of adoption for *every* constant defined in the JavaCard specification.

Figure 5 shows the popularity of both symmetric and asymmetric cryptography signature schemes. The ECDSA signature algorithm, together with RSA with PKCS#1 padding, is the most popular, followed by the 3DES-based and HMAC-based MAC schemes. Relatively few applets utilize other options. Related Figure 6 shows the adoption of different variants of hash functions in combination with ECDSA over time, and Figure 10 compares time of introduction and use of two frequently used signature algorithms – `ALG_RSA_SHA_PKCS1` and `ALG_ECDSA_SHA_256`.

The popular key lengths are shown in Figure 7 and they are mostly as expected – AES-128/256b and 3DES-112/168b are used frequently for symmetric cryptography, EC-FP-256b and RSA-2048b for asymmetric one. The only surprise is still a high usage of RSA-1024b. Related Figure 8 demonstrates the trend for DES algorithm with one (`LENGTH_DES`), two (`LENGTH_DES3_2KEY`) and three (`LENGTH_DES3_3KEY`) keys in time, showing adoption of longer keys later.

The usage of 3rd-party, sometimes proprietary, APIs is shown in Figure 9. GlobalPlatform/OpenPlatform API is used mostly for the handling of secure channel messages or accessing global card state. If proprietary API is used (NXP, Giesecke+Devrient), the applet can be statically analyzed for the usage of JavaCard features but cannot be dynamically analyzed due to the missing proprietary SDK. The results for other relevant classes are available in supplementary materials[14].

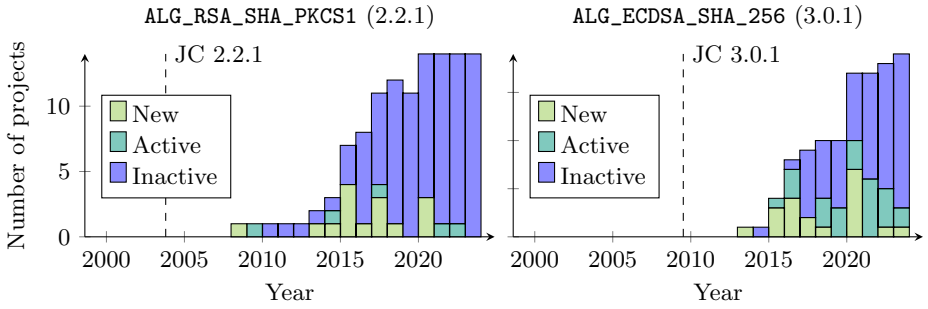

Fig. 10. Comparison of the adoption of `ALG_RSA_SHA_PKCS1` and `ALG_ECDSA_SHA_256` constants. The ECDSA signature was adopted only in 2013 and is still being added to the source code of active projects, while the RSA signature was included already since 2008, but overall less frequently and not after the year 2020.

We also analyzed the adoption of different constants from JavaCard API up to 3.2 with 583 values in total. A surprisingly high number of 404 constants are completely unused (69.3%), and 488 constants are used in less than six projects (83.7%). Only 42 constants are used in more than 25 (7.2%) projects. Table 4 provides complete statistics.

Table 4. The usage of constants from JavaCard API by the set of all applets.

# applets using constant	no use	1	2-5	6-10	11-25	26-50	51-75	76-100	100+
# API constants (583)	404	36	48	20	33	21	7	4	10

[14] `https://crocs.fi.muni.cz/papers/cardis2023`

4.2 Memory requirements analysis

The memory requirements by a given applet can be inferred from the difference in reported available memory right after the applet's constructor method is entered and just before it is finished. Automatic code instrumentation using Spoon library [11] was used to insert the corresponding memory measurements methods `JCSystem.getAvailableMemory()` into every analyzed applet.

When an instrumented code line is reached, the instrumented applet stores the amount of free memory for all memory types at the given point in time (persistent, transient reset, and transient deselect), due to the limitations of the `JCSystem.getAvailableMemory()` JavaCard API, the gathered values are capped at 32 kB (3.0.3 and older) or 2 GB (3.0.4 and newer). Hence, usage of smartcards with JavaCard 3.0.4 or newer is preferable for analysis of applet's persistent memory. After the applet instance creation is finished, all measurements are transferred to the host controller using an additionally added custom command. Note that the method described measures not only memory consumption by allocated primitive arrays but also transient and persistent memory consumed by instances of cryptographic objects (keys, engines) created.

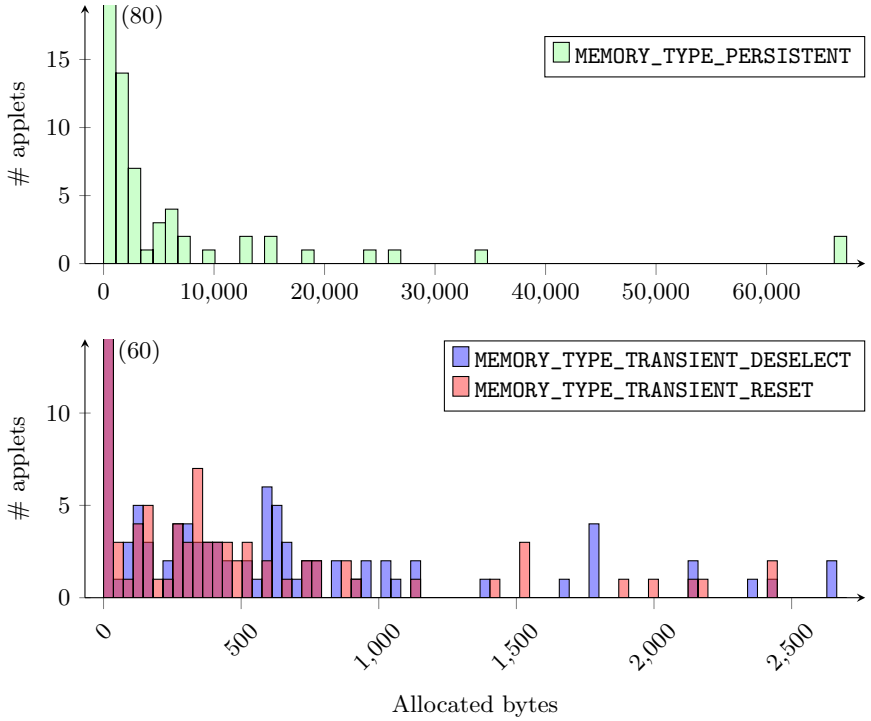

Fig. 11. Histograms of memory allocation in entry point class constructors per applet on NXP JCOP4 J3R180 with maximum available transient memory around 3.8 kB. Note that the first largest bin is clipped in both graphs.

The evaluation was performed on three physical cards and was restricted only to the 144 applets that were previously successfully tested using the jCardSim simulator in Section 3.4, which is the minimum prerequisite for successful installation on the physical cards. All three selected cards support at least JavaCard API 3.0.4 and thus have more precise memory measurements.

The representative results for the NXP JCOP4 J3R180 card with the most applets installable (124 in total) are shown in Figure 11 for the transient and persistent memory types, respectively. The results obtained for other cards were closely matching these results with only small differences. The differences are caused by slightly different memory requirements of cryptographic objects on different cards. The large majority of applets require less than 1 kB of transient memory, with 59 applets requiring less than 70 bytes. Only a handful of applets required more than 2 kB of transient memory, notably the project eVerification-2[15] (2.6 kB).

Note that we cannot detect applets requiring *more* memory than available on the target smartcard using this methodology as the installation will fail, and the measured memory values are not retrieved. We have manually analyzed the 20 applets, which failed to install and assessed the reason for failure, and found out that nine applets likely failed due to transient memory requirements larger than what is available on the JCOP4 card.

Similarly, a large majority of applets required less than 10 kB of persistent memory, with only ten applets requiring more, notably applets from the PinSentry-OTP[16] project (67 kB) and the smart_card_TLS[17] project (34 kB). Twenty-seven applets required only 100 B or less of persistent memory.

5 Discussion and limitations

JavaCard's open-source ecosystem is relatively long-running, with the first commits made already in the year 2004, although only infrequently. It became sharply more active from the year 2013, reaching more than 30 active projects (code changes done to actual JavaCard code) in the years 2015, 2017, and 2020 (maximum of 44 in 2015), although with significant variability between the years. However, there seems to be a significant decline after the year 2020, with only 18 and 16 projects active in years 2021 and 2022, respectively, and only 7 in the whole first half of the year 2023. We do not have a good explanation for this trend, especially given the increased supply of smartcards available in small quantities, which are often used by open-source developers. It will be interesting to observe if the trend confirms in coming years. A decrease in the development activity of the JavaCard open-source ecosystem may be only temporary if the open-source ecosystem is generally copying the trends in the certification ecosystem. The certification activity increased in 2009, peaked in 2013 (32 certificates), and experienced a local minimum in 2019 (12 certificates issued), increasing again since then. Open-source ecosystem gained activity similarly, but with a delay of about four years, so we may see growth in activity again soon.

New algorithms from JavaCard specification are slow to be adopted. It takes at least two and up to five years for smartcards with support for the algorithms

[15] `https://github.com/CRISES-URV/eVerification-2`.
[16] `https://github.com/Celliwig/PinSentry-OTP`.
[17] `https://github.com/gilb/smart_card_TLS`.

introduced in the new version of the JavaCard specification to be certified. An additional delay of an algorithm used in open-source implementations is caused by the unavailability of freshly certified cards in small quantities to open-source developers. Even if it is reasonable to assume that commercial closed-source applets may adopt a new algorithm sooner as pre-certification testing sample smartcards may be available, the widespread use of an implementation based on a not-yet-certified smartcard is unlikely.

Only a small subset of around 30% of algorithms and constants defined in the JavaCard specification is used by open-source applets and it consists mostly of the common ones – SHA1 and SHA256 for hashing, RSA 1024/2048b and EC FP 256b for signing, AES128/256b, 3DES and RSA with PKCS1 or no padding for the encryption.

The open-source cryptographic implementations (outside JavaCard domain) tend to adopt new algorithms sooner than their closed-source counterparts serving more established and, thus, slower-moving domains. An example might be the adoption of Curve25519 and related algorithms like Ed25519 or X25519, which are long-time popular in more open domains yet only recently become considered for adoption in mainstream domains like government-recognized signatures or signatures of PDF documents in Adobe Reader. Similarly, JavaCard open-source projects would likely utilize Curve25519 and related algorithms but were not able to due to missing direct support for these features (optional support for `X25519` and `Ed25519` was introduced only in JavaCard API 3.1). As Section 2 demonstrates, only a handful of products for API version 3.1 are now certified and likely still not available to open-source developers. As implementation of these algorithms requires access to cryptographic co-processors, compensating around missing features is not easy or even possible. For example, open-source implementation of Ed25519[18] become available only in 2022 and is based on Curve25519 implementation[19] utilizing host-side computation and JCMathLib library [8]; having non-trivial RAM requirements and not providing a high level of security against side-channel attacks due to higher leakage of JCVM in comparison to fully native implementation. We can conclude that the JavaCard open-source ecosystem is significantly held back by the slow introduction of new features in the specification and further by their availability in actual physical smartcards. As a result, the whole smartcard ecosystem is likely negatively impacted by limited exposure to new ideas and usage scenarios, serving only well-established domains and harming its long-term competitiveness. The gradual replacement of functionality once delivered by smartcards by software-only implementations is not caused only by easier deployment but also by the inability of smartcards to deliver the functionality required.

Algorithms tend to stay in the source code once included as typical example in Figure 10 demonstrates and is observed for almost all constants used. As a result, the legacy algorithms typically continue to be supported by smart-

[18] `https://github.com/dufkan/JCEd25519`.
[19] `https://github.com/david-oswald/jc_curve25519`.

cards, although they may be discouraged from use (e.g., single DES or MD5) to maintain backward compatibility.

The open-source applets usually do not depend on the proprietary extension packages as their use is typically limited under a non-disclosure agreement (NDA) which would prevent public availability of the applet's source code. The exceptions are GlobalPlatform, SIM toolkit, and ThothTrust KM101 packages where NDA is not required. Despite NDA limitations, NXP and G+D proprietary packages are referenced from publicly available code, although without export files necessary for the cap file conversion.

Memory requirements tend to be low due to the simplicity of applet tasks and to fit into restrictions of widely available cards. Open-source applets tend to utilize less than 1 kB of transient memory (RAM) and less than 10 kB of persistent memory (EEPROM/flash). No measured applet uses more than 2.7 kB of RAM and 70 kB of EEPROM. The exceptions are applets trying to implement advanced features otherwise unavailable on cards like Ed25519 or multiparty signature algorithms like FROST [4].

5.1 Known limitations

The analysis performed in this paper has its limitations, mainly stemming from the limited access to non-public closed-source applets and detailed proprietary documents about certified products.

The final date of the specification release is not the date when the specification becomes available for the first time to the smartcard manufacturers, as manufacturers are typically involved in the draft preparation of the new release. The actual span for adoption from the first availability of a given feature by a specific card is, therefore, likely longer than reported in this paper. Similarly, the date of the certification is not the date when the card becomes available to potential customers. Engineering preview samples or production-ready but not-yet-certified cards might be available earlier. This decreases the time span between the availability of the given feature by a specific card.

An algorithm might be available sooner in the proprietary packages provided by a vendor than standardized in the public JavaCard API. Closed-source applets may adopt features earlier than open-source ones or keep using the proprietary variants after the feature is available in the public API.

Only open-source applets are analyzed. The closed-source proprietary code may exhibit different usage patterns and are likely to use proprietary extension packages more frequently. However, the usage will still be limited by the algorithms available on the underlying platform as partially mapped by the JCAlgTest project [12] and certification analysis from this paper in Section 2.

Memory consumption for closed-source applets may be smaller than for some more complex open-source ones, especially when features unavailable in the public API are required. The prime example might be the utilization of the JCMathLib library [8] with relatively large RAM requirements to expose lower-level `ECPoint` operations, which are easily available in proprietary packages with almost no overhead.

We tested the deployability of applets on physical smartcards only up to the `select()` method. The applet functionality may still systematically fail later, even if provided with correct input data, for example, if the required engine is not allocated in the applet's constructor, but only when the functionality triggering command is received. However, such a situation shall be relatively uncommon as a good programming practice in the JavaCard world is to allocate all required resources in the applet's constructor. The full examination would require the usage of the host-side controlling application, which is frequently not available.

Some algorithms from certification documents may be missed due to PDF parsing errors, and not all supported algorithms may be explicitly listed in the smartcard certificate. As the total number of JavaCard-related certificates is not overwhelming (59 in total), we checked all the certificates for possible PDF-to-text errors while we cannot compensate for the omitted ones.

6 Conclusions

The paper primarily provides data-based insight into the adoption rate of features introduced by the JavaCard specification versions using security certification reports and a large database of open-source applets.

The analysis of JavaCard-related constants from certification artifacts shows increased certification activity since the year 2006, with around twenty certified products every year for the whole last decade and taking 1-2 years before the first certified product implements at least part of the newly released specification. However, the speed of adoption of its features is relatively slow, typically six or more years before a feature is widely supported by certified products. Around 70% of the features listed are never or only marginally supported.

The static and dynamic analysis of JavaCard open-source projects shows increased activity from the year 2013 but possibly declining during the last two years. Around 20 projects achieved wider popularity and development activity. The rate of adoption of new specification features is slower, likely correlated with the unavailability of recent performant smartcards in smaller quantities, which also causes limited applet portability between different available physical cards. The specification features tend to stay in the existing source code, making feature deprecation more complicated, especially when a feature intended to replace the deprecated one is not supported by the majority of cards. The open-source applets typically require only a small amount of transient and persistent memory, with some notable exceptions. The applets with large transient memory requirements are typically the ones that need to complicatedly compensate for the unavailability of some low-level features like `ECPoint` operations in the public API. JavaCard open-source ecosystem is likely held back by the slow introduction of new features into the specification and further delayed by their inaccessibility of physical smartcards with desired algorithmic support.

Acknowledgments: We would like to thank reviewers for their valuable comments. The authors were supported by Ai-SecTools (VJ02010010) project.

References

1. Olavo Barbosa and Carina Alves. A systematic mapping study on software ecosystems. *Proc. Int'l Workshop on Soft. Ecos*, 2011.
2. Licel Corporation. jCardSim — Java Card Runtime Environment Simulator. `https://jcardsim.org/`, 2022. Accessed: 2023-09-29.
3. Awdren de Lima Fontao, Rodrigo Pereira dos Santos, and Arilo Claudio Dias-Neto. Mobile software ecosystem (mseco): a systematic mapping study. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 653–658. IEEE, 2015.
4. Antonín Dufka and Petr Švenda. Enabling efficient threshold signature computation via Java Card API. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery.
5. John Glossner, Samantha Murphy, and Daniel Iancu. An overview of the drone open-source ecosystem. *arXiv preprint arXiv:2110.02260*, 2021.
6. Jan Hajny, Lukas Malina, Zdenek Martinasek, and Ondrej Tethal. Performance evaluation of primitives for privacy-enhancing cryptography on current smart-cards and smart-phones. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 17–33. Springer, 2014.
7. Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems–a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.
8. Vasilios Mavroudis and Petr Svenda. JCMathLib: Wrapper cryptographic library for transparent and certifiable javacard applets. In *2020 IEEE European Symposium on Security and Privacy Workshops*, pages 89–96, Genoa, Italy, 2020. IEEE.
9. Oracle. Java card™ platform, application programming interface, classic edition version 3.2. `https://docs.oracle.com/en/java/javacard/3.2/jcapi/api_classic/index.html`, 2023. Accessed: 2023-09-29.
10. Martin Paljak. GlobalPlatformPro. `https://github.com/martinpaljak/GlobalPlatformPro`, 2023. Accessed: 2023-09-29.
11. Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.
12. Petr Svenda, Rudolf Kvasnovsky, Imrich Nagy, and Antonin Dufka. JCAlgTest: Robust identification metadata for certified smartcards. In *19th International Conference on Security and Cryptography*, pages 597–604, Lisabon, 2022. INSTICC.

Appendix

JavaCard projects and certificates included in analysis

The following types of JavaCard open-source projects were included in the analysis[20]: Electronic passports and citizen ID (8x), Authentication and access control (29x), Payments and loyalty (20x), Key and password managers (15x), Digital signing, OpenPGP and mail security (8x), e-Health (1x), NDEF tags (6x), Cryptocurrency wallets (7x), Emulation of proprietary cards (8x), Mobile telephony (SIM) (5x), Library JavaCard code (47x), Learning (school projects, etc.) (5x) and Other (24x).

Table 5. Common Criteria and FIPS certificates used for algorithm analysis.

5G PK 5.2.2 Advanced SIM (D00233151F016B)	https://seccerts.org/cc/cb872473562ed2cf/
Athena IDPass ICAO BAC avec AA sur composant SB23YR48/80B avec librairie cryptographique NesLib v3.0	https://seccerts.org/cc/d49988efd778ca9d/
Athena IDPass ICAO EAC avec AA sur composant SB23YR48/80B avec librairie cryptographique NesLib v3.0	https://seccerts.org/cc/337ece90615ed69d/
Athena IDProtect Duo v10 (in BAC Configuration)	https://seccerts.org/cc/598e2a9978d3c79c/
Athena IDProtect Duo v10 (in EAC Configuration)	https://seccerts.org/cc/4285d9b580f8a2a6/
Athena IDProtect Duo v5 avec application IASECC en configuration ICAO BAC sur composant AT90SC28880RCFV	https://seccerts.org/cc/16b24ff1bf3c079b/
Athena IDProtect Duo v5 avec application IASECC en configuration ICAO EAC sur composant AT90SC28880RCFV	https://seccerts.org/cc/e23f9c02819688f6/
Athena IDProtect Duo v5 avec applicationIASECC en configuration ICAO BAC surcomposant AT90SC28880RCFV	https://seccerts.org/cc/797ea41cc2e95f08/
Athena IDProtect Duo v5 avec applicationIASECC en configuration ICAO EAC surcomposant AT90SC28880RCFV	https://seccerts.org/cc/a49706f436a68734/
Athena IDProtect/OS755 (release 0355, level 0602, correctif P6) avec application IAS-ECC sur composants SB23YR48/80B	https://seccerts.org/cc/b168db2651cbc1d6/
Athena IDProtect/OS755 (release 0355, level0802, correctif P8) avec application IAS-ECC surcomposants SB23YR48/80B	https://seccerts.org/cc/be9ca77bec616fe5/
Athena IDProtect/OS755 (release 4016, level0101) avec application IAS-ECC sur composantsSB23YR48/80B	https://seccerts.org/cc/3cb89a7fc7672364/
Athena IDProtect/OS755 Key version 9.1.2 on AT90SC25672RCT-USB Microcontroller embedding IDSign applet	https://seccerts.org/cc/82dce1546c693694/
Athena IDProtect/OS755 avec application IAS-ECC sur composants SB23YR48/80B	https://seccerts.org/cc/79f485411f64e16ff/
Athena IDProtect/OS755 avec application ICAO BAC sur composants SB23YR48/80B	https://seccerts.org/cc/ba521df2c728c5a2/
Athena IDProtect/OS755 avec application ICAO EAC sur composants SB23YR48/80B	https://seccerts.org/cc/c8fc487eec95c21e/
Athena OS755/IDProtect v6 avec application IAS-ECC sur composant AT90SC28872RCU	https://seccerts.org/cc/fca98ecd003e1b82/
Carte UpTeq NFC3.2.2_Generic v1.0 sur composant ST33G1M2-F	https://seccerts.org/cc/60f0dd83c8f32b8c/
ID-One Cosmo 32 v5	https://seccerts.org/fips/fcf1e4bf9cc9c108/
ID-One Cosmo 64 v5	https://seccerts.org/fips/068814875fcdfb8f/
ID-One Cosmo 64 v5	https://seccerts.org/fips/4a30a219a9d7663f/
ID-One Cosmo 64 v5	https://seccerts.org/fips/6e2d4d3a3d4c4bb2/
IDeal Citiz v2.1 Open platform	https://seccerts.org/cc/f67f3737a3b47208/
IDeal Citiz v2.1 STC Open Platform	https://seccerts.org/cc/05c7e778528a1596/
IDeal Citiz v2.1.1 Open platform on M7892 B11	https://seccerts.org/cc/8782c7c292ef2759/
IDeal Citiz v2.1.1 Open platform on M7893 B11	https://seccerts.org/cc/29b0321ec6b75ebd/
IDeal Citiz v2.15-i on Infineon M7892 B11 Java Card Open Platform	https://seccerts.org/cc/c301c38902477230/
IDeal Citiz v2.16-i on M7892 B11 - Java Card Open Platform	https://seccerts.org/cc/4dc023ea2e3c4115/
IDeal Citiz v2.17-i on Infineon M7892 B11 Java Card Open Platform	https://seccerts.org/cc/08408cbc394e7421/
IDeal Citiz v2.17-i on Infineon M7893 B11 Java Card Open Platform	https://seccerts.org/cc/40fa0a4cb5193977/
Infineon SECORA™ ID S v1.1 (SLJ52GxxyyyzS)	https://seccerts.org/cc/8c92d4b773f61ca8/
Infineon SECORA™ ID X v1.1 (SLJ52GxAyyyzX)	https://seccerts.org/cc/cd4d1f03b2d3b1cc/
MultiApp v4.0.1 with Filter Set 1.0 Java Card Open Platform on M7892 G12 chip	https://seccerts.org/cc/dcb259767b98f381/
NXP JAVA OS1 ChipDoc v1.0 SSCD (J3K080/J2K080)	https://seccerts.org/cc/30120e4f3aa2f30a/
NXP JCOP 3 EMV P60	https://seccerts.org/cc/61832cb4291c343f/
NXP JCOP 3 P60	https://seccerts.org/cc/3feda0b8b5637540/
NXP JCOP 3 SECID P60 (OSA)	https://seccerts.org/cc/d699ed2b1adc6be4/
NXP JCOP 3 SECID P60 (OSA) PL2/5	https://seccerts.org/cc/3e08a27e9d9c9b1e/
NXP JCOP 4 P71	https://seccerts.org/cc/3d8083b1e6c7b336/
NXP JCOP 4 SE050M	https://seccerts.org/cc/5a71cce42550635e/
NXP JCOP 4.0 on P73N2M0	https://seccerts.org/cc/173704f0d2b8a02f/
NXP JCOP 4.5 P71	https://seccerts.org/cc/f0d94de8beabbd84/
NXP JCOP 4.7 SE051	https://seccerts.org/cc/6cd7a7a1cffaa67e/
NXP JCOP 4.x on P73N2M0B0.2C2/2C6 Secure Element	https://seccerts.org/cc/5f12d355f8acb7cd/
NXP JCOP 5.1 on SN100.C48 Secure Element	https://seccerts.org/cc/c8982f2e8de39b22/
NXP JCOP 5.2 on SN100.C58 Secure Element	https://seccerts.org/cc/4526aa14337a2b0c/
NXP JCOP 6.2 on SN220 Secure Element, R1.01.1, R1.02.1, R1.02.1-1, R2.01.1	https://seccerts.org/cc/3b0d05e2fe06c803/
NXP JCOP 7.0 on SN300 Secure Element, JCOP 7.0 R1.62.0.1	https://seccerts.org/cc/b0e6f667d52402df/
NXP JCOP 7.0 with eUICC extension on SN300 Secure Element, JCOP 7.0 R1.64.0.2	https://seccerts.org/cc/45098872448f5816/
NXP JCOP on SN100.C25 Secure Element	https://seccerts.org/cc/03aded94fb04c62e/
NXP JCOP6.x on SN200.C04 Secure Element	https://seccerts.org/cc/ae175aa839dbf692/
Oberthur ID-One Cosmo 128 v5.5 D	https://seccerts.org/fips/6d094db49a6e2242/
Oberthur ID-One Cosmo 128 v5.5 for DoD CAC	https://seccerts.org/fips/a5bef651c8e3fd6c/
Plateforme JavaCard MultiApp V4.0.1 - PACE en configuration ouverte masquée sur le composant M7892 G12	https://seccerts.org/cc/40fc6ad0aed92913/
SafeNet eToken version 9.1.2 Athena IDProtect/OS755 on INSIDE Secure AT90SC25672RCTUSB embedding IDSign applet	https://seccerts.org/cc/87a80325171d8add/
SafeNet eToken - Athena IDProtect/OS755 on Atmel AT90SC25672RCT-USB embedding IDSign applet	https://seccerts.org/cc/c3d0eacd0639efb5/
Thales NFC422 v1.0 JCS	https://seccerts.org/cc/e38c6956b53dfd36/
Thales TESS v3.0 Platform	https://seccerts.org/cc/9be0c86102117436/
XSmart OpenPlatform V1.1 on S3CT9KW/S3CT9KC/S3CT9K9	https://seccerts.org/cc/cb203cf5d91b1ae3/

The JavaCard version reference analysis included all certification documents from the dataset containing a match of the JavaCard API version using regular expressions (338 documents in total).

[20] https://github.com/crocs-muni/javacard-curated-list