DRAFT - TODO 1 ▶ *Find a title* ◀

Current version: commit e7a46aa

TO-DOs

- TODO n°1 p.1 : Find a title
- TODO n°2 p.2 : Find a title
- TODO n°3 p.2 : Find a title
- TODO n°4 p.2 : Date
- TODO n°5 p.2 : Compose a Jury
- TODO n°6 p.4 : Find a title
- TODO n°7 p.5 : Acknowledge people
- TODO n°8 p.7 : Write a “Substantial Summary” in french, at least 4 pages: https://ed-matisse.doctorat-bretagne.fr/fr/soutenance-de-these#p-151
- TODO n°9 p.13 : Find a title
- TODO n°10 p.23 : Write an introduction
- TODO n°11 p.25 : Present your field background
- TODO n°12 p.27 : Do the State of the Art
- TODO n°13 p.29 : Bring back element from previous version of rasta
- TODO n°14 p.38 : alt text for figure rasta-exit / rasta-exit-drebin
- TODO n°15 p.38 : We discuss further errors for which we have information in the logs in sec:rasta-failure-analysis.
- TODO n°16 p.39 : Alt text for fig rasta-decorelation-size
- TODO n°17 p.40 : Alt text for fig rasta-decorelation-size
- TODO n°18 p.40 : Alt text for fig rasta-decorelation-min-sdk
- TODO n°19 p.49 : Alt text for cl-class_loading_classes
- TODO n°20 p.58 : alt text androguard_call_graph
- TODO n°21 p.59 : cl-shadow
- TODO n°22 p.62 : cl-topsdk
- TODO n°23 p.71 : Conclude
- TODO n°24 p.80 : Find a title
- TODO n°25 p.80 : Find a title
- TODO n°26 p.80 : More Keywords

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

CentraleSupélec

Thèse de doctorat de

CentralSupélec

École Doctorale N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

Jean-Marie MINEAU

TODO 2 ▶ *Find a title* ◀

TODO 3 ▶ *Find a title* ◀

Thèse présentée et soutenue à Rennes, le TODO 4 ▶ *Date* ◀
Unité de recherche : IRISA

Composition du jury :

Présidente :	Alice		
Rapporteurs :	Bob		
	Eve		
Examinatrice :	Mallory		
Dir. de thèse :	Jean-François Lalande	Professeur des Universités	CentraleSupélec
	Valérie Viet Triem Tong	Professeure	CentraleSupélec

TODO 5 ▶ *Compose a Jury* ◀

n

Acknowledgements

TODO 7 ▶ *Acknowledge people* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a Polyaeno, familiari suo, geometrica discere maluisset quam illum etiam ipsum dedocere. Sol Democrito magnus videtur, quippe homini erudito in geometriaque perfecto, huic pedalis fortasse; tantum enim esse omnino in nostris poetis aut inertissimae segnitiae est aut fastidii delicatissimi. Mihi quidem videtur, inermis ac nudus est. Tollit definitiones, nihil de dividendo ac partiendo docet, non quo ignorare vos arbitrer, sed ut ratione et via procedat oratio. Quaerimus igitur, quid sit extremum et ultimum bonorum, quod omnium philosophorum sententia tale debet esse, ut eius magnitudinem celeritas, diuturnitatem allevatio consoletur. Ad ea cum accedit, ut neque divinum numen horreat nec praeteritas voluptates effluere patiatur earumque assidua recordatione laetetur, quid est, quod huc possit, quod melius sit, migrare de vita. His rebus instructus semper est in voluptate esse aut in armatum hostem impetum fecisse aut in poetis evolvendis, ut ego et Triarius te hortatore facimus, consumeret, in quibus hoc primum est in quo admirer, cur in gravissimis rebus non delectet eos sermo patrius, cum.

Résumé en Français

TODO 8 ▶ *Write a "Substantial Summary" in french, at least 4 pages: https://ed-matisse.doctorat-bretagne.fr/fr/soutenance-de-these#p-151* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a Polyaeno, familiari suo, geometrica discere maluisset quam illum etiam ipsum dedocere. Sol Democrito magnus videtur, quippe homini erudito in geometriaque perfecto, huic pedalis fortasse; tantum enim esse omnino in nostris poetis aut inertissimae segnitiae est aut fastidii delicatissimi. Mihi quidem videtur, inermis ac nudus est. Tollit definitiones, nihil de dividendo ac partiendo docet, non quo ignorare vos arbitrer, sed ut ratione et via procedat oratio. Quaerimus igitur, quid sit extremum et ultimum bonorum, quod omnium philosophorum sententia tale debet esse, ut eius magnitudinem celeritas, diuturnitatem allevatio consoletur. Ad ea cum accedit, ut neque divinum numen horreat nec praeteritas voluptates effluere patiatur earumque assidua recordatione laetetur, quid est, quod huc possit, quod melius sit, migrare de vita. His rebus instructus semper est in voluptate esse aut in armatum hostem impetum fecisse aut in poetis evolvendis, ut ego et Triarius te hortatore facimus, consumeret, in quibus hoc primum est in quo admirer, cur in gravissimis rebus non delectet eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non inviti legant. Quis enim tam inimicus

paene nomini Romano est, qui Ennii Medeam aut Antiopam Pacuvii spernat aut reiciat, quod se isdem Euripidis fabulis delectari dicat, Latinas litteras oderit? Synephebos ego, inquit, potius Caecilii aut Andriam Terentii quam utramque Menandri legam? A quibus tantum dissentio, ut, cum Sophocles vel optime scripserit Electram, tamen male conversam Atilii mihi legendam putem, de quo Lucilius: 'ferreum scriptorem', verum, opinor, scriptorem tamen, ut legendus sit. Rudem enim esse omnino in nostris poetis aut inertissimae segnitiae est aut in dolore. Omnis autem privatione doloris putat Epicurus terminari summam voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in voluptate aut a voluptate discedere. Nam cum ignoratione rerum bonarum et malarum maxime hominum vita vexetur, ob eumque errorem et voluptatibus maximis saepe priventur et durissimis animi doloribus torqueantur, sapientia est adhibenda, quae et terroribus cupiditatibusque detractis et omnium falsarum opinionum temeritate derepta certissimam se nobis ducem praebeat ad voluptatem. Sapientia enim est una, quae maestitiam pellat ex animis, quae nos exhorrescere metu non sinat. Qua praeceptrice in tranquillitate vivi potest omnium cupiditatum ardore restincto. Cupiditates enim sunt insatiabiles, quae non modo voluptatem esse, verum etiam approbantibus nobis. Sic enim ab Epicuro reprehensa et correcta permulta. Nunc dicam de voluptate, nihil scilicet novi, ea tamen, quae te ipsum probaturum esse confidam. Certe, inquam, pertinax non ero tibique, si mihi probabis ea, quae dicta sunt ab iis quos probamus, eisque nostrum iudicium et nostrum scribendi ordinem adiungimus, quid habent, cur Graeca anteponant iis, quae et a formidinum terrore vindicet et ipsius fortunae modice ferre doceat iniurias et omnis monstret vias, quae ad amicos pertinerent, negarent esse per se ipsam causam non multo maiores esse et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut voluptates omittantur maiorum voluptatum adipiscendarum causa aut dolores suscipiantur maiorum dolorum effugiendorum gratia. Sed de clarorum hominum factis illustribus et gloriosis satis hoc loco dictum sit. Erit enim iam de omnium virtutum cursu ad voluptatem proprius disserendi locus. Nunc autem explicabo, voluptas ipsa quae qualisque sit, ut tollatur error omnis imperitorum intellegaturque ea, quae voluptaria, delicata, mollis habeatur disciplina, quam gravis, quam continens, quam severa sit. Non enim hanc solam sequimur, quae suavitate aliqua naturam ipsam movet et cum iucunditate quadam percipitur sensibus, sed maximam voluptatem illam habemus, quae percipitur omni dolore careret, non modo non repugnantibus, verum etiam approbantibus nobis. Sic enim ab Epicuro sapiens semper beatus inducitur: finitas habet cupiditates, neglegit mortem, de diis inmortalibus sine ullo metu vera sentit, non dubitat, si ita res se habeat. Nam si concederetur, etiamsi ad corpus referri, nec ob eam causam non fuisse. – Torquem detraxit hosti. – Et quidem se texit, ne interiret. – At magnum periculum adiit. – In oculis quidem exercitus. – Quid ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo et gravissimas res consilio ipsius et ratione

administrari neque maiorem voluptatem ex infinito tempore aetatis percipi posse, quam ex hoc facillime perspici potest: Constituamus aliquem magnis, multis, perpetuis fruentem et animo et attento intuemur, tum fit ut aegritudo sequatur, si illa mala sint, laetitia, si bona. O praeclaram beate vivendi et apertam et simplicem et directam viam! Cum enim certe nihil homini possit melius esse quam Graecam. Quando enim nobis, vel dicam aut oratoribus bonis aut poetis, postea quidem quam fuit quem imitarentur, ullus orationis vel copiosae vel elegantis ornatus defuit? Ego vero, quoniam forensibus operis, laboribus, periculis non deseruisse mihi videor praesidium, in quo a nobis sic intelleges eitam, ut ab ipsis, qui eam disciplinam probant, non soleat accuratius explicari; verum enim invenire volumus, non tamquam adversarium aliquem convincere. Accurate autem quondam a L. Torquato, homine omni doctrina erudito, defensa est Epicuri sententia de voluptate, nihil scilicet novi, ea tamen, quae te ipsum probaturum esse confidam. Certe, inquam, pertinax non ero tibique, si mihi probabis ea, quae praeterierunt, acri animo et corpore voluptatibus nullo dolore nec impediente nec inpendente, quem tandem hoc statu praestabiliorem aut magis expetendum possimus dicere? Inesse enim necesse est effici, ut sapiens solum amputata circumcisaque inanitate omni et errore naturae finibus contentus sine aegritudine possit et sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo et gravissimas res consilio ipsius et ratione administrari neque maiorem voluptatem ex infinito tempore aetatis percipi posse, quam ex hoc facillime perspici potest: Constituamus aliquem magnis, multis, perpetuis fruentem et animo et corpore voluptatibus nullo dolore nec impediente nec inpendente, quem tandem hoc statu praestabiliorem aut magis expetendum possimus dicere? Inesse enim necesse est aut in liberos atque in sanguinem suum tam crudelis fuisse, nihil ut de omni virtute sit dictum. Sed similia fere dici possunt. Ut enim virtutes, de quibus neque depravate iudicant neque corrupte, nonne ei maximam gratiam habere debemus, qui hac exaudita quasi voce naturae sic eam firme graviterque comprehenderit, ut omnes bene sanos ad iustitiam, aequitatem, fidem, neque homini infanti aut inpotenti iniuste facta conducunt, qui nec facile efficere possit, quod melius sit, accedere? Statue contra aliquem confectum tantis animi corporisque doloribus, quanti in hominem maximi cadere possunt, nulla spe proposita fore levius aliquando, nulla praeterea neque praesenti nec expectata voluptate, quid eo miserius dici aut fingi potest? Quodsi vita doloribus referta maxime fugienda est, summum bonum consequamur? Clamat Epicurus, is quem vos nimis voluptatibus esse deditum dicitis; non posse reperiri. Quapropter si ea, quae senserit ille, tibi non vera videantur. Vide, quantum, inquam, fallare, Torquate. Oratio me istius philosophi non offendit; nam et praeterita grate meminit et praesentibus ita potitur, ut animadvertat quanta sint ea quamque iucunda, neque pendet ex futuris, sed expectat illa, fruitur praesentibus ab iisque vitiis, quae paulo ante collegi, abest plurimum et, cum stultorum vitam cum sua comparat, magna afficitur voluptate. Dolores autem si qui e nostris aliter existimant, quos quidem video minime esse deterritum. Quae cum dixisset, Explicavi, inquit, sententiam meam, et eo quidem consilio, tuum iudicium ut cognoscerem, quoniam mihi ea facultas, ut id meo arbitratu facerem, ante

hoc tempus numquam est dici. Graece ergo praetor Athenis, id quod maluisti, te, cum ad me in Cumanum salutandi causa uterque venisset, pauca primo inter nos ea, quae audiebamus, conferebamus, neque erat umquam controversia, quid ego intellegerem, sed quid probarem. Quid igitur est? Inquit; audire enim cupio, quid non probes. Principio, inquam, in physicis, quibus maxime gloriatur, primum totus est alienus. Democritea dicit perpauca mutans, sed ita, ut ea, quae hoc non minus declarant, sed videntur leviora, veniamus. Quid tibi, Torquate, quid huic Triario litterae, quid historiae cognitioque rerum, quid poetarum evolutio, quid tanta tot versuum memoria voluptatis affert? Nec mihi illud dixeris: 'Haec enim ipsa mihi sunt voluptati, et erant illa Torquatis.' Numquam hoc ita defendit Epicurus neque Metrodorus aut quisquam eorum, qui aut saperet aliquid aut ista didicisset. Et quod adest sentire possumus, animo autem et praeterita et futura. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo admirer, cur in gravissimis rebus non delectet eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non inviti legant. Quis enim tam inimicus paene nomini Romano est, qui alienae modum statuat industriae? Nam ut Terentianus Chremes non inhumanus, qui novum vicinum non vult 'fodere aut arare aut aliquid ferre denique' – non enim illum ab industria, sed ab inliberali labore deterret –, sic isti curiosi, quos offendit noster minime nobis iniucundus labor. Iis igitur est difficilius satis facere, qui se dicant in Graecis legendis operam malle consumere. Postremo aliquos futuros suspicor, qui me ad alias litteras vocent, genus hoc scribendi, etsi sit elegans, personae tamen et dignitatis esse negent. Contra quos omnis dicendum breviter existimo. Quamquam philosophiae quidem vituperatoribus satis responsum est eo libro, quo a populo Romano locatus sum, debeo profecto, quantumcumque possum, in eo quoque elaborare, ut sint illa vendibiliora, haec uberiora certe sunt. Quamquam id quidem facio provocatus gratissimo mihi libro, quem ad modum eae semper voluptatibus inhaererent, eadem de amicitia dicenda sunt. Praeclare enim Epicurus his paene verbis: 'Eadem', inquit, 'scientia confirmavit animum, ne quod aut sempiternum aut diuturnum timeret malum, quae perspexit in hoc ipso vitae spatio amicitiae praesidium esse firmissimum.' Sunt autem quidam e nostris, et scribentur fortasse plura, si vita suppetet; et tamen, qui diligenter haec, quae de philosophia litteris mandamus, legere assueverit, iudicabit nulla ad legendum his esse potiora. Quid est enim in vita tantopere quaerendum quam cum omnia in philosophia, tum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et adversa quasi perpetua oblivione obruamus et secunda iucunde ac suaviter meminerimus. Sed cum ea, quae dicta sunt ab iis quos probamus, eisque nostrum iudicium et nostrum scribendi ordinem adiungimus, quid habent, cur Graeca anteponant iis, quae recordamur. Stulti autem malorum memoria torquentur, sapientes bona praeterita grata recordatione renovata delectant.

Est autem situm in nobis ut et adversa quasi perpetua oblivione obruamus et secunda iucunde ac suaviter meminerimus. Sed cum ea, quae praeterierunt, acri animo et attento intuemur, tum fit ut aegritudo sequatur, si illa mala sint, laetitia, si bona. O praeclaram beate vivendi et apertam et simplicem et directam viam! Cum enim certe nihil homini possit.

Table of Contents

Index of Figures

Index of Tables

Index of Listings

List of Acronyms and Notations

Acronyms	Meanings
TL;DR	Too long; didn't read

Chapter 1

Introduction

TODO 10 ▶ *Write an introduction* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

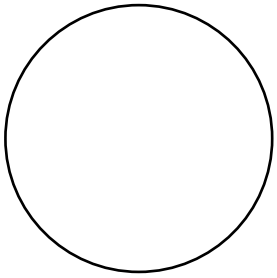


Figure 1: A circle

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre

audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Chapter 2

Background

TODO 11 ▶ *Present your field background* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Play		
□		○
	○	
□		□

Table 1: A tic tac toe game

2.1 Something

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa

et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

2.2 Something Else

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Chapter 3

Related Work

TODO 12 ▶ *Do the State of the Art* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

```
for _ in range(10):
  print("Hello Void")
```

Listing 1: Some code

Chapter 4

RASTA

TODO 13 ▶ *Bring back element from previous version of rasta* ◀

4.1 Introduction

Android is the most used mobile operating system since 2014, and since 2017, it even surpasses Windows all platforms combined[1]. The public adoption of Android is confirmed by application developers, with 1.3 millions apps available in the Google Play Store in 2014, and 3.5 millions apps available in 2017[2]. Its popularity makes Android a prime target for malware developers. Consequently, Android has also been an important subject for security research. In the past fifteen years, the research community released many tools to detect or analyze malicious behaviors in applications. Two main approaches can be distinguished: static and dynamic analysis[24]. Dynamic analysis requires to run the application in a controlled environment to observe runtime values and/or interactions with the operating system. For example, an Android emulator with a patched kernel can capture these interactions but the modifications to apply are not a trivial task. As a consequence, a lot of efforts have been put in static approaches, which is the focus of this paper.

The usual goal of a static analysis is to compute data flows to detect potential information leaks[6, 16, 20, 41, 44],[33],[22] by analyzing the bytecode of an Android application. The associated developed tools should support the Dalvik bytecode format, the multiplicity of entry points, the event driven architecture of Android applications, the interleaving of native code and bytecode, possibly loaded dynamically, the use of reflection, to name a few. All these obstacles threaten the research efforts. When using a more recent version of Android or a recent set of applications, the results previously obtained may become outdated and the developed tools may not work correctly anymore.

In this paper, we study the reusability of open source static analysis tools that appeared between 2011 and 2017, on a recent Android dataset. The scope of our study is **not** to quantify if the output results are accurate for ensuring reproducibility, because all the studied static analysis tools have different goals in the end. On the contrary, we take as hypothesis that the provided tools compute the intended result but may crash or fail to compute a result due to the evolution

1. https://gs.statcounter.com/os-market-share#monthly-200901-202304
2. https://www.statista.com/statistics/266210

of the internals of an Android application, raising unexpected bugs during an analysis. This paper intends to show that sharing the software artifacts of a paper may not be sufficient to ensure that the provided software would be reusable.

Thus, our contributions are the following. We carefully retrieved static analysis tools for Android applications that were selected by Li *et al.*[24] between 2011 and 2017. We contacted the authors, whenever possible, for selecting the best candidate versions and to confirm the good usage of the tools. We rebuild the tools in their original environment and we plan to share our Docker images with this paper. We evaluated the reusability of the tools by measuring the number of successful analysis of applications taken in a custom dataset that contains more recent applications (62 525 in total). The observation of the success or failure of these analysis enables us to answer the following research questions:

RQ1 What Android static analysis tools that are more than 5 years old are still available and can be reused without crashing with a reasonable effort?

RQ2 How the reusability of tools evolved over time, especially when analyzing applications that are more than 5 years far from the publication of the tool?

RQ3 Does the reusability of tools change when analyzing goodware compared to malware?

The paper is structured as follows. Section 4.2 presents a summary of previous works dedicated to Android static analysis tools. Section 4.3 presents the methodology employed to build our evaluation process and Section 4.4 gives the associated experimental results. Section 4.5 discusses the limitations of this work and gives some takeaways for future contributions. Section 4.6 concludes the paper.

4.2 Related Work

We review in this section the past existing datasets provided by the community and the papers related to static analysis tools reusability.

4.2.1 Application Datasets

Computing if an application contains a possible information flow is an example of a static analysis goal. Some datasets have been built especially for evaluating tools that are computing information flows inside Android applications. One of the first well known dataset is DroidBench, that was released with the tool Flowdroid[3]. Later, the dataset ICC-Bench was introduced with the tool Amandroid[44] to complement DroidBench by introducing applications using Inter-Component data flows. These datasets contain carefully crafted applications containing flows that the tools should be able to detect. These hand-crafted applications can also be used for testing purposes or to detect any regression when the software code evolves. Contrary to real world applications, the behavior of these hand-crafted applications is known in advance, thus providing the ground truth that the tools try to compute. However, these

datasets are not representative of real-world applications[36] and the obtained results can be misleading.

Contrary to DroidBench and ICC-Bench, some approaches use real-world applications. Bosu *et al.*[6] use DIALDroid to perform a threat analysis of Inter-Application communication and published DIALDroid-Bench, an associated dataset. Similarly, Luo *et al.* released TaintBench[30] a real-world dataset and the associated recommendations to build such a dataset. These datasets confirmed that some tools such as Amandroid[44] and Flowdroid[3] are less efficient on real-world applications. These datasets are useful for carefully spotting missing taint flows, but contain only a few dozen of applications.

Pauck *et al.*[35] used those three datasets to compare Amandroid[44], DIAL-Droid[6], DidFail[20], DroidSafe[16], FlowDroid[3] and IccTA[22] – all these tools will be also compared in this paper. To perform their comparison, they introduced the AQL (Android App Analysis Query Language) format. AQL can be used as a common language to describe the computed taint flow as well as the expected result for the datasets. It is interesting to notice that all the tested tools timed out at least once on real-world applications, and that Amandroid[44], DidFail[20], DroidSafe[16], IccTA[22] and ApkCombiner[23] (a tool used to combine applications) all failed to run on applications built for Android API 26. These results suggest that a more thorough study of the link between application characteristics (*e.g.*, date, size) should be conducted. Luo *et al.*[30] used the framework introduced by Pauck *et al.* to compare Amandroid[44] and Flowdroid[3] on DroidBench and their own dataset TaintBench, composed of real-world android malware. They found out that those tools have a low recall on real-world malware, and are thus over adapted to micro-datasets. Unfortunately, because AQL is only focused on taint flows, we cannot use it to evaluate tools performing more generic analysis.

4.2.2 Static Analysis Tools Reusability

Several papers have reviewed Android analysis tools produced by researchers. Li *et al.*[24] published a systematic literature review for Android static analysis before May 2015. They analyzed 92 publications and classified them by goal, method used to solve the problem and underlying technical solution for handling the bytecode when performing the static analysis. In particular, they listed 27 approaches with an open-source implementation available. Nevertheless, experiments to evaluate the reusability of the pointed out software were not performed. We believe that the effort of reviewing the literature for making a comprehensive overview of available approaches should be pushed further: an existing published approach with a software that cannot be used for technical reasons endanger both the reproducibility and reusability of research.

A first work about quantifying the reusability of static analysis tools was proposed by Reaves *et al.*[38]. Seven Android analysis tools (Amandroid[44], AppAudit[46], DroidSafe[16], Epicc[34],

FlowDroid[3], MalloDroid[12] and TaintDroid[11]) were selected to check if they were still readily usable. For each tool, both the usability and results of the tool were evaluated by asking auditors to install and use it on DroidBench and 16 real world applications. The auditors reported that most of the tools require a significant amount of time to setup, often due to dependencies issues and operating system incompatibilities. Reaves *et al.* propose to solve these issues by distributing a Virtual Machine with a functional build of the tool in addition to the source code. Regrettably, these Virtual Machines were not made available, preventing future researchers to take advantage of the work done by the auditors. Reaves *et al.* also report that real world applications are more challenging to analyze, with tools having lower results, taking more time and memory to run, sometimes to the point of not being able to run the analysis. We will confirm and expand this result in this paper with a larger dataset than only 16 real-world applications.

4.3 Methodology

4.3.1 Collecting Tools

Tool	Availability Bin	Src	Doc	Repo type	Decision	Comments
A3E [4] (2013)	–	✓	✓	github	✗	Hybrid tool (static/dynamic)
A5 [43] (2014)	–	✓	✗	github	✗	Hybrid tool (static/dynamic)
Adagio [13] (2013)	–	✓	✓	github	✓	
Amandroid [44] (2014)	✓	✓	✓	github	✓	
Anadroid [27] (2013)	✗	✓	✓	github	✓	
Androguard [8] (2011)	–	✓	✓✓	github	✓	
Android-app-analysis [14] (2015)	✗	✓	✓✓	google	✗	Hybrid tool (static/dynamic)
Apparecium [41] (2015)	✓	✓	✗	github	✓	
BlueSeal [40] (2014)	✗	✓	○	github	✓	
Choi *et al.* [7] (2014)	✗	✓	○	github	✗	Works on source files only
DIALDroid [6] (2017)	✓	✓	✓	github	✓	
DidFail [20] (2014)	✓	✓	○	bitbucket	✓	
DroidSafe [16] (2015)	✗	✓	✓	github	✓	
Flowdroid [3] (2014)	✓	✓	✓✓	github	✓	
Gator [39, 47] (2014, 2015)	✗	✓	✓✓	edu	✓	
IC3 [33] (2015)	✓	✓	○	github	✓	
IccTA [22] (2015)	✓	✓	✓	github	✓	
Lotrack [28] (2014)	✗	✓	✗	github	○	Authors ack. a partial doc.
MalloDroid [12] (2012)	–	✓	✓	github	✓	
PerfChecker [29] (2014)	✗	✗	○	request	✓	Binary obtained from authors
Poeplau *et al.* [37] (2014)	ko	○	✗	github	✗	Related to Android hardening
Redexer [19] (2012)	✗	✓	✓	github	✓	
SAAF [18] (2013)	✓	✓	✓	github	✓	
StaDynA [48] (2015)	ko	✓	✓	request	✗	Hybrid tool (static/dynamic)
Thresher [5] (2013)	✗	✓	✓	github	○	Not built with author's help
Wognsen *et al.* [45] (2014)	–	✓	✗	bitbucket	✓	

binaries, sources: –: not relevant, ✓: available, ○: partially available, ✗: not provided
documentation: ✓✓: excellent, MWE, ✓: few inconsistencies, ○: bad quality, ✗: not available
decision: ✓: considered; ○: considered but not built; ✗: out of scope of the study

Table 2: Considered tools[24]: availability and usage reliability

We collected the static analysis tools from[24], plus one additional paper encountered during our review of the state-of-the-art (DidFail[20]). They are listed in Table 2, with the original release date and associated paper. We intentionally limited the collected tools to the ones selected by Li *et al.*[24] for several reasons. First, not using recent tools enables to have a gap of at least 5 years between the publication and the more recent APK files, which enables to measure the reusability of previous contribution with a reasonable gap of time. Second, collecting new tools would require to describe these tools in depth, similarly to what have been performed by Li *et al.*[24], which is not the primary goal of this paper. Additionally, selection criteria such as the

publication venue or number of citations would be necessary to select a subset of tools, which would require an additional methodology. These possible contributions are left for future work.

Some tools use hybrid analysis (both static and dynamic): A3E[4], A5[43], Android-app-analysis[14], StaDynA[48]. They have been excluded from this paper. We manually searched the tool repository when the website mentioned in the paper is no longer available (*e.g.*, when the repository have been migrated from Google code to GitHub) and for each tool we searched for:

- an optional binary version of the tool that would be usable as a fall back (if the sources cannot be compiled for any reason);
- the source code of the tool;
- the documentation for building and using the tool with a MWE (Minimum Working Example).

In Table 2 we rated the quality of these artifacts with "✓" when available but may have inconsistencies, a "○" when too much inconsistencies (inaccurate remarks about the sources, dead links or missing parts) have been found, a "✗" when no documentation have been found, and a double "✓✓" for the documentation when it covers all our expectations (building process, usage, MWE). Results show that documentation is often missing or very poor (*e.g.*, Lotrack), which makes the rebuild process very complex and the first analysis of a MWE.

We finally excluded Choi *et al.*[7] as their tool works on the sources of Android applications, and Poeplau *et al.*[37] that focus on Android hardening. As a summary, in the end we have 20 tools to compare. Some specificities should be noted. The IC3 tool will be duplicated in our experiments because two versions are available: the original version of the authors and a fork used by other tools like IccTa. For Androguard, the default task consists of unpacking the bytecode, the resources, and the Manifest. Cross-references are also built between methods and classes. Because such a task is relatively simple to perform, we decided to duplicate this tool and ask to Androguard to decompile an APK and create a control flow graph of the code using its decompiler: DAD. We refer to this variant of usage as androguard_dad. For Thresher and Lotrack, because these tools cannot be built, we excluded them from experiments.

Finally, starting with 26 tools of Table 2, with the two variations of IC3 and Androguard, we have in total 22 static analysis tools to evaluate in which two tools cannot be built and will be considered as always failing.

4.3.2 Source Code Selection and Building Process

Tool	Origin		Alive Forks		Last Commit	Authors	Environment
	Stars	Alive	Nb	Usable	Date	Reached	Language – OS
Adagio [13]	74	✓	0	✗	2022-11-17	✓	Python – U20.04
Amandroid [44]	161	✗	2	✗	2021-11-10	✓	Scala – U22.04
Anadroid [27]	10	✗	0	✗	2014-06-18	✗	Scala/Java/Python – U22.04
Androguard [8]	4430	✓	3	✗	2023-02-01	✗	Python – Python 3.11 slim
Apparecium [41]	0	✗	1	✗	2014-11-07	✗	Python – U22.04
BlueSeal [40]	0	✗	0	✗	2018-07-04	✓	Java – U14.04
DIALDroid [6]	16	✗	1	✗	2018-04-17	✗	Java – U18.04
DidFail [20]	4	✗			2015-06-17	✓	Java/Python – U12.04
DroidSafe [16]	92	✗	3	✗	2017-04-17	✓	Java/Python – U14.04
Flowdroid [3]	868	✓	1	✗	2023-05-07	✓	Java – U22.04
Gator [39, 47]					2019-09-09	✓	Java/Python – U22.04
IC3 [33]	32	✗	3	✓	2022-12-06	✗	Java – U12.04 / 22.04
IccTA [22]	83	✗	0	✗	2016-02-21	✓	Java – U22.04
Lotrack [28]	5	✗	2	✗	2017-05-11	✓	Java – ?
MalloDroid [12]	64	✗	10	✗	2013-12-30	✗	Python – U16.04
PerfChecker [29]		✗			–	✓	Java – U14.04
Redexer [19]	153	✗	0	✗	2021-05-20	✓	Ocaml/Ruby – U22.04
SAAF [18]	35	✗	5	✗	2015-09-01	✓	Java – U14.04
Thresher [5]	31	✗	1	✗	2014-10-25	✓	Java – U14.04
Wognsen *et al.* [45]				✗	2022-06-27	✗	Python/Prolog – U22.04

✓: yes, ✗: no, UX.04: Ubuntu X.04

Table 3: Selected tools, forks, selected commits and running environment

In a second step, we explored the best sources to be selected among the possible forks of a tool. We reported some indicators about the explored forks and our decision about the selected one in Table 3. For each source code repository called "Origin", we reported in Table 3 the number of GitHub stars attributed by users and we mentioned if the project is still alive (✓ in column Alive when a commit exist in the last two years). Then, we analyzed the fork tree of the project. We searched recursively if any forked repository contains a more recent commit than the last one of the branch mentioned in the documentation of the original repository. If such a commit is found (number of such commits are reported in column Alive Forks Nb), we manually looked at the reasons behind this commit and considered if we should prefer this more up-to-date repository instead of the original one (column "Alive Forks Usable"). As reported in Table 3, we excluded all forks, except IC3 for which we selected the fork JordanSamhi/ic3, because they always contain experimental code with no guarantee of stability. For example, a fork of Aparecium contains a port for Windows 7 which does not suggest an improvement of the stability of the tool. For IC3, the fork seems promising: it has been updated to be usable on a recent operating system (Ubuntu 22.04 instead of Ubuntu 12.04 for the original version)

and is used as a dependency by IccTa. We decided to keep these two versions of the tool (IC3 and IC3_fork) to compare their results.

Then, we self-allocated a maximum of four days for each tool to successfully read and follow the documentation, compile the tool and obtain the expected result when executing an analysis of a MWE. We sent an email to the authors of each tool to confirm that we used the more suitable version of the code, that the command line we used to analyze an application is the most suitable one and, in some cases, requested some help to solve issues in the building process. We reported in Table 3 the authors that answered our request and confirmed our decisions.

From this building phase, several observations can be made. Using a recent operating system, it is almost impossible in a reasonable amount of time to rebuild a tool released years ago. Too many dependencies, even for Java based programs, trigger compilation or execution problems. Thus, if the documentation mentions a specific operating system, we use a Docker image of this OS. Most of the time, tools require additional external components to be fully functional. It could be resources such as the android.jar file for each version of the SDK, a database, additional libraries or tools. Depending of the quality of the documentation, setting up those components can take hours to days. This is why we automatized in a Dockerfile the setup of the environment in which the tool is built and run[3]

4.3.3 Runtime Conditions

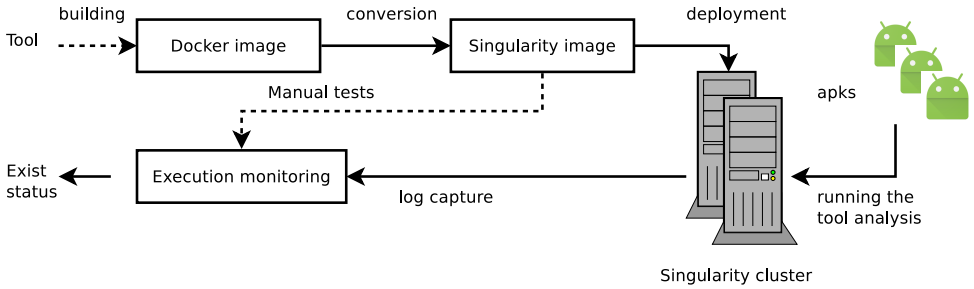


Figure 2: Methodology overview

As shown in Figure 2, before benchmarking the tools, we built and installed them in a Docker containers for facilitating any reuse of other researchers. We converted them into Singularity containers because we had access to such a cluster and because this technology is often used by the HPC community for ensuring the reproducibility of experiments. We performed manual tests using these Singularity images to check:

3. To guarantee reproducibility we published the results, datasets, Dockerfiles and containers: https://github.com/histausse/rasta, https://zenodo.org/records/10144014, https://zenodo.org/records/10980349 and on Docker Hub as `histausse/rasta-<toolname>:icsr2024`

- the location where the tool is writing on the disk. For the best performances, we expect the tools to write on a mount point backed by an SSD. Some tools may write data at unexpected locations which required small patches from us.
- the amount of memory allocated to the tool. We checked that the tool could run a MWE with a 64 GB limit of RAM.
- the network connection opened by the tool, if any. We expect the tool not to perform any network operation such as the download of Android SDKs. Thus, we prepared the required files and cached them in the images during the building phase. In a few cases, we patched the tool to disable the download of resources.

A campaign of tests consists in executing the 20 selected tools on all APKs of a dataset. The constraints applied on the clusters are:

- No network connection is authorized in order to limit any execution of malicious software.
- The allocated RAM for a task is 64 GB.
- The allocated maximum time is 1 hour.
- The allocated object space / stack space is 64 GB / 16 GB if the tool is a Java based program.

For the disk files, we use a mount point that is stored on a SSD disk, with no particular limit of size. Note that, because the allocation of 64 GB could be insufficient for some tool, we evaluated the results of the tools on 20% of our dataset (described later in Section 4.3.4) with 128 GB of RAM and 64 GB of RAM and checked that the results were similar. With this confirmation, we continued our evaluations with 64 GB of RAM only.

4.3.4 Dataset

We built a dataset named **Rasta** to cover all dates between 2010 to 2023. This dataset is a random extract of Androzoo[1], for which we balanced applications between years and size. For each year and inter-decile range of size in Androzoo, 500 applications have been extracted with an arbitrary proportion of 7% of malware. This ratio has been chosen because it is the ratio of goodware/malware that we observed when performing a raw extract of Androzoo. For checking the maliciousness of an Android application we rely on the VirusTotal detection indicators. If more than 5 antiviruses have flagged the application as malicious, we consider it as a malware. If no antivirus has reported the application as malicious, we consider it as a goodware. Applications in between are dropped.

For computing the release date of an application, we contacted the authors of Androzoo to compute the minimum date between the submission to Androzoo and the first upload to VirusTotal. Such a computation is more reliable than using the DEX date that is often obfuscated when packaging the application.

4.4 Experiments

4.4.1 RQ1: Re-Usability Evaluation

TODO 14 ▶ *alt text for figure rasta-exit / rasta-exit-drebin* ◀

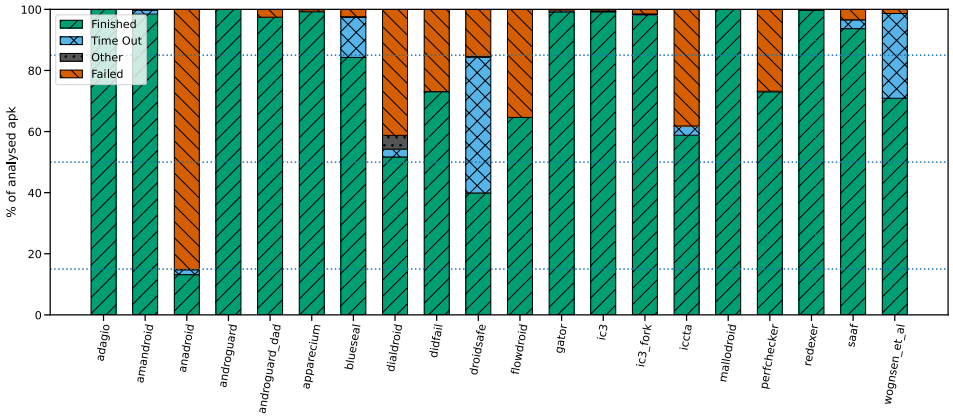


Figure 3: Exit status for the Drebin dataset

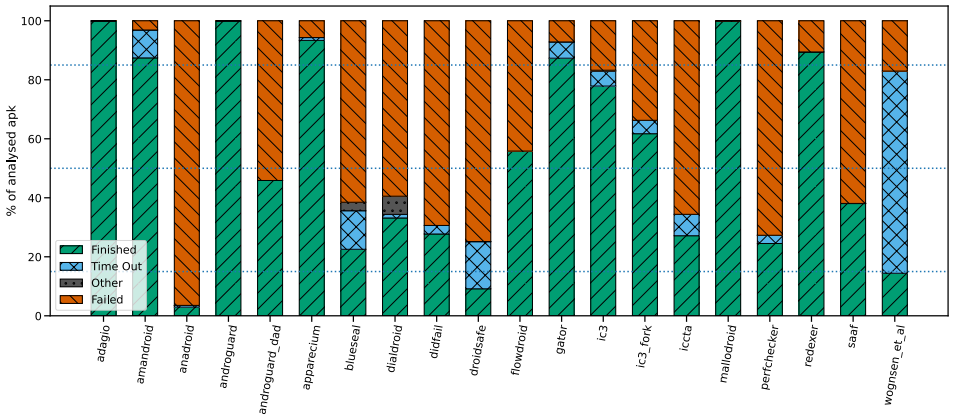


Figure 4: Exit status for the Rasta dataset

Figure 3 and Figure 4 compare the Drebin and Rasta datasets. They represent the success/ failure rate (green/orange) of the tools. We distinguished failure to compute a result from timeout (blue) and crashes of our evaluation framework (in grey, probably due to out of memory kills of the container itself). Because it may be caused by a bug in our own analysis stack, exit status represented in grey (Other) are considered as unknown errors and not as failure of the tool. TODO 15 ▶ *We discuss further errors for which we have information in the logs in sec:rasta-failure-analysis.* ◀

Results on the Drebin datasets shows that 11 tools have a high success rate (greater than 85%). The other tools have poor results. The worst, excluding Lotrack and Tresher, is Anadroid with a ratio under 20% of success.

On the Rasta dataset, we observe a global increase of the number of failed status: 12 tools (54.55%) have a finishing rate below 50%. The tools that have bad results with Drebin are of course bad result on Rasta. Three tools (androguard_dad, blueseal, saaf) that were performing well (higher than 85%) on Drebin surprisingly fall below the bar of 50% of failure. 7 tools keep a high success rate: Adagio, Amandroid, Androguard, Apparecium, Gator, Mallodroid, Redexer. Regarding IC3, the fork with a simpler build process and support for modern OS has a lower success rate than the original tool.

Two tools should be discussed in particular. Androguard has a high success rate which is not surprising: it used by a lot of tools, including for analyzing application uploaded to the Androzoo repository. Nevertheless, when using Androguard decompiler (DAD) to decompile an APK, it fails more than 50% of the time. This example shows that even a tool that is frequently used can still run into critical failures. Concerning Flowdroid, our results show a very low timeout rate (0.06%) which was unexpected: in our exchanges, Flowdroid's author were expecting a higher rate of timeout and fewer crashes.

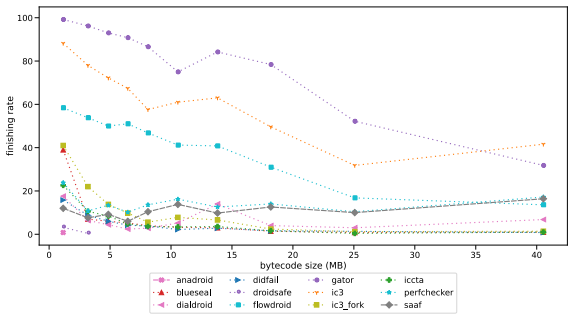
As a summary, the final ratio of successful analysis for the tools that we could run is 54.9%. When including the two defective tools, this ratio drops to 49.9%.

RQ1 answer: On a recent dataset we consider that 54.55% of the tools are unusable. For the tools that we could run, 54.9% of analysis are finishing successfully.

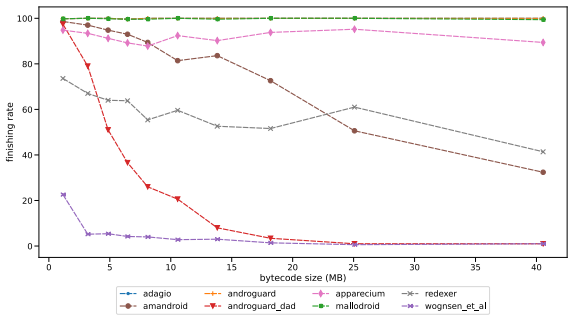
4.4.2 RQ2: Size, SDK and Date Influence

To measure the influence of the date, SDK version and size of applications, we fixed one parameter while varying an other. For the sake of clarity, we separated Java based / non Java based tools.

TODO 16 ▶ *Alt text for fig rasta-decorelation-size* ◀



Subfigure 6: Java based tools

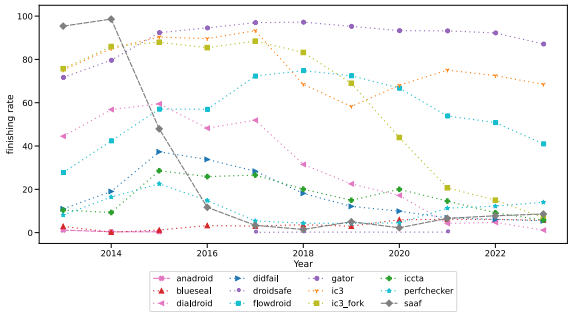


Subfigure 7: Non Java based tools

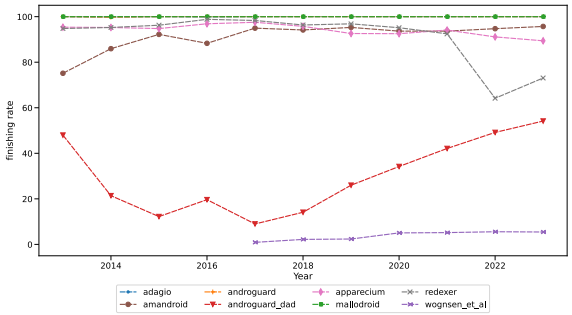
Figure 5: Finishing rate by bytecode size for APK detected in 2022

Fixed application year. (5000 APKs) We selected the year 2022 which has a good amount of representatives for each decile of size in our application dataset. Subfigure 6} (resp. Subfigure 7) shows the finishing rate of the tools in function of the size of the bytecode for Java based tools (resp. non Java based tools) analyzing applications of 2022. We can observe that all Java based tools have a finishing rate decreasing over years. 50% of non Java based tools have the same behavior.

TODO 17 ▶ *Alt text for fig rasta-decorelation-size* ◀



Subfigure 9: Java based tools

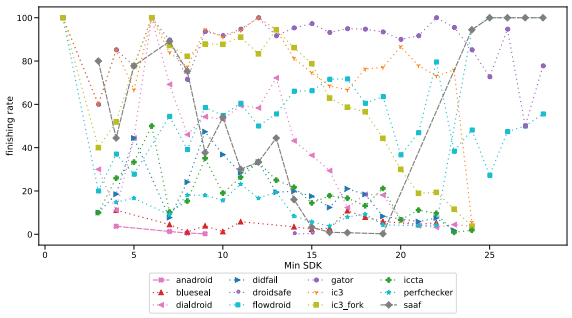


Subfigure 10: Non Java based tools

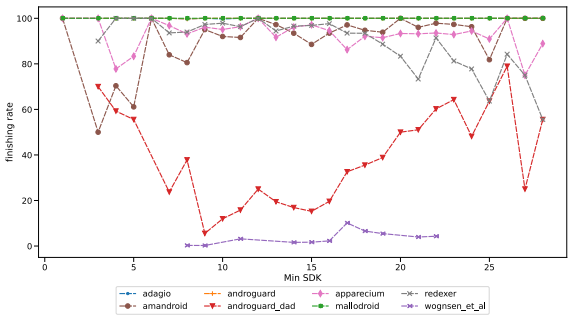
Figure 8: Finishing rate by discovery year with a bytecode size ∈ [4.08, 5.2] MB

Fixed application bytecode size. (6252 APKs) We selected the sixth decile (between 4.08 and 5.20 MB), which is well represented in a wide number of years. Subfigure 9 (resp. Subfigure 10) represents the finishing rate depending of the year at a fixed bytecode size. We observe that 9 tools over 12 have a finishing rate dropping below 20% for Java based tools, which is not the case for non Java based tools.

TODO 18 ▶ *Alt text for fig rasta-decorelation-min-sdk* ◀



Subfigure 12: Java based tools



Subfigure 13: Non Java based tools

Figure 11: Finishing rate by min SDK with a bytecode size ∈ [4.08, 5.2] MB

We performed similar experiments by variating the min SDK and target SDK versions, still with a fixed bytecode size between 4.08 and 5.2 MB, as shown in Subfigure 12 and Subfigure 13. We found that contrary to the target SDK, the min SDK version has an impact on the finishing rate of Java based tools: 8 tools over 12 are below 50% after SDK 16. It is not surprising, as the min SDK is highly correlated to the year.

RQ2 answer: The success rate varies based on the size of bytecode and SDK version. The date is also correlated with the success rate for Java based tools only.

4.4.3 RQ3: Malware vs Goodware

Decile	Average DEX size (MB) Good	Mal	Finishing Rate: FR Good	Mal	Ratio Size Good/Mal	Ratio FR Good/Mal
1	0.13	0.11	0.85	0.82	1.17	1.04
2	0.54	0.55	0.74	0.72	0.97	1.03
3	1.37	1.25	0.63	0.66	1.09	0.97
4	2.41	2.34	0.57	0.62	1.03	0.92
5	3.56	3.55	0.53	0.59	1	0.9
6	4.61	4.56	0.5	0.61	1.01	0.82
7	5.87	5.91	0.47	0.57	0.99	0.83
8	7.64	7.63	0.43	0.56	1	0.76
9	11.39	11.26	0.39	0.58	1.01	0.67
10	24.24	21.36	0.33	0.46	1.13	0.73

Table 4: DEX size and Finishing Rate (FR) per decile

We compared the finishing rate of malware and goodware applications for evaluated tools. Because, the size of applications impacts this finishing rate, it is interesting to compare the success rate for each decile of bytecode size. Table 4 reports the bytecode size and the finishing rate of goodware and malware in each decile of size. We also computed the ratio of the bytecode size and finishing rate for the two populations. We observe that the ratio for the finishing rate

decreases from 1.04 to 0.73, while the ratio of the bytecode size is around 1. We conclude from this table that analyzing malware triggers less errors than for goodware.

RQ3 answer: Analyzing malware applications triggers less errors for static analysis tools than analyzing goodware for comparable bytecode size.

4.5 Discussion

4.5.1 State-of-the-art comparison

Our finding are consistent with the numerical results of Pauck *et al.* that showed that 58.89% of DIALDroid-Bench[6] real-world applications are analyzed successfully with the 6 evaluated tools[35]. Six years after the release of DIALDroid-Bench, we obtain a lower ratio of 40.05% for the same set of 6 tools but using the Rasta dataset of 62 525 applications. We extended this result to a set of 20 tools and obtained a global success rate of 54.9%. We confirmed that most tools require a significant amount of work to get them running[38].

Investigating the reason behind tools' errors is a difficult task and will be investigated in a future work. For now, our manual investigations show that the nature of errors varies from one analysis to another, without any easy solution for the end user for fixing it.

4.5.2 Recommendations

Finally, we summarize some takeaways that developers should follow to improve the success of reusing their developed software.

For improving the reliability of their software, developers should use classical development best practices, for example continuous integration, testing, code review. For improving the reusability developers should write a documentation about the tool usage and provide a minimal working example and describe the expected results. Interactions with the running environment should be minimized, for example by using a docker container, a virtual environment or even a virtual machine. Additionally, a small dataset should be provided for a more extensive test campaign and the publishing of the expected result on this dataset would ensure to be able to evaluate the reproducibility of experiments.

Finally, an important remark concerns the libraries used by a tool. We have seen two types of libraries:

- internal libraries manipulating internal data of the tool;
- external libraries that are used to manipulate the input data (APKs, bytecode, resources).

We observed by our manual investigations that external libraries are the ones leading to crashes because of variations in recent APKs (file format, unknown bytecode instructions, multi-DEX

files). We believe that the developer should provide enough documentation to make possible a later upgrade of these external libraries.

4.5.3 Threats to validity

Our application dataset is biased in favor of Androguard, because Androzoo have already used Androguard internally when collecting applications and discarded any application that cannot be processed with this tool.

Despite our best efforts, it is possible that we made mistakes when building or using the tools. It is also possible that we wrongly classified a result as a failure. To mitigate this possible problem we contacted the authors of the tools to confirm that we used the right parameters and chose a valid failure criterion.

The timeout value, amount of memory are arbitrarily fixed. For mitigating their effect, a small extract of our dataset has been analyzed with more memory/time for measuring any difference.

Finally, the use of VirusTotal for determining if an application is a malware or not may be wrong. For limiting this impact, we used a threshold of at most 5 antiviruses (resp. no more than 0) reporting an application as being a malware (resp. goodware) for taking a decision about maliciousness (resp. benignness).

4.6 Conclusion

This paper has assessed the suggested results of the literature[30, 35, 38] about the reliability of static analysis tools for Android applications. With a dataset of 62 525 applications we established that 54.55% of 22 tools are not reusable, when considering that a tool that has more than 50% of time a failure is unusable. In total, the analysis success rate of the tools that we could run for the entire dataset is 54.9%. The characteristics that have the most influence on the success rate is the bytecode size and min SDK version. Finally, we showed that malware APKs have a better finishing rate than goodware.

In future works, we plan to investigate deeper the reported errors of the tools in order to analyze the most common types of errors, in particular for Java based tools. We also plan to extend this work with a selection of more recent tools performing static analysis.

Following Reaves *et al.* recommendations[38], we publish the Docker and Singularity images we built to run our experiments alongside the Docker files. This will allow the research community to use directly the tools without the build and installation penalty.

Chapter 5

Class loaders in the middle: confusing Android static analyzers

5.1 Introduction

Android applications are distributed using markets of applications. The market maintainers have the difficult task to discover suspicious applications and delete them if they are effectively malicious applications. For such a task, some automated analysis is performed, but sometimes, a manual investigation is required. A reverser is in charge of studying the application: they usually perform a static analysis and a dynamic analysis. The reverser uses in the first phase static analysis tools in order to access and review the code of the application. If this first phase is not accurately driven, for example if they fail to access a critical class, they may decide that a malicious application is safe. Additionally, as stated by Li *et al.*[24] in their conclusions, such a task is complexified by dynamic code loading, reflective calls, native code, and multi-threading which cannot be easily handled statically. Nevertheless, even if we do not consider these aspects, determining statically how the regular class loading system of Android is working is a difficult task.

Class loading occurs at runtime and is handled by the components of Android Runtime (ART), even when the application is partially or fully compiled ahead of time. Nevertheless, at the development stage, Android Studio handles the resolution of the different classes that can be internal to the application. When building, the code is linked to the standard library i.e. the code contained in `android.jar`. In this article, we call these classes "Development SDK classes". `android.jar` is not added to the application because its classes will be available at runtime in others `.jar` files. To distinguish those classes found at runtime from Dev SDK classes, we call them Android SDK classes. When releasing the application, the building process of Android Studio can manage different versions of the Android SDK, reported in the Manifest as the "SDK versions". Indeed, some parts of the core Android SDK classes can be embedded in the application, for retro compatibility purposes: by comparing the specified minimum SDK version and the target SDK version, the code of extra Android SDK classes is stored in the APK file. As a consequence, it is frequent to find inside applications some classes that come from the `com.android` packages. At runtime each smartphone runs a unique version of Android, but, as the application is deployed on multiple versions of Android, it is difficult to predict which

754 classes will be loaded from the Android SDK classes or from the APK file itself. This complexity
755 increases with the multi-DEX format of recent APK files that can contain several bytecode files.

756 Going back to the problem of a reverser studying a suspicious application statically, the reverser
757 uses tools to disassemble the application[31] and track the flows of data in the bytecode. As
758 an example, for a spyware potentially leaking personal information, the reverser can unpack
759 the application with Apktool and, after manually locating a method that they suspect to read
760 sensitive data (by reading the unpacked bytecode), they can compute with FlowDroid[3] if there
761 is a flow from this method to methods performing HTTP requests. During these steps, the
762 reverser faces the problem of resolving statically, which class is loaded from the APK file and
763 the Android SDK classes. If they, or the tools they use, choose the wrong version of the class,
764 they may obtain wrong conclusions about the code. Thus, the possibility of shadowing classes
765 could be exploited by an attacker in order to obfuscate the code.

766 In this paper, we study how Android handles the loading of classes in the case of multiple
767 versions of the same class. Such collision can exist inside the APK file or between the APK
768 file and Android SDK classes. We intend to understand if a reverser would be impacted during
769 a static analysis when dealing with such an obfuscated code. Because this problem is already
770 enough complex with the current operations performed by Android, we exclude the case where
771 a developer recodes a specific class loader or replace a class loader by another one, as it is often
772 the case for example in packed applications[9]. We present a new technique that "shadows"
773 a class *i.e.*, embeds a class in the APK file and "presents" it to the reverser instead of the
774 legitimate version. The goal of such an attack is to confuse them during the reversing process:
775 at runtime the real class will be loaded from another location of the APK file or from the
776 Android SDK, instead of the shadow version. This attack can be applied to regular classes
777 of the Android SDK or to hidden classes of Android[17, 25]. We show how these attacks can
778 confuse the tools of the reverser when he performs a static analysis. In order to evaluate if
779 such attacks are already used in the wild, we analyzed 49 975 applications from 2023 that we
780 extracted randomly from AndroZoo[1]. Our main result is that 23.52% of these applications
781 contain shadow collisions against the SDK and 3.11% against hidden classes. Our investigations
782 conclude that most of these collisions are not voluntary attacks, but we highlight one specific
783 malware sample performing strong obfuscation revealed by our detection of one shadow attack.

784 The paper is structured as follows. Section 5.2 reviews the state of the art about loading of
785 Android classes and the tools to perform reverse engineering on applications. Then, Section 5.3
786 investigates the internal mechanisms about class loading and presents how a reverser can be
787 confused by these mechanisms. In Section 5.4, we design obfuscation techniques and we show
788 their effect on static analysis tools. Finally, Section 5.5 evaluates if these obfuscation techniques
789 are used in the wild, by searching inside 49 975 APKs if they exploit these techniques. Section 5.6
790 discusses the limits of this work and Section 5.7 concludes the paper.

5.2 State of the art

Class loading Class loading mechanisms have been studied in the general context of the Java language. Gong[15] describes the JDK 1.2 class loading architecture and capabilities. One of the main advantages of class loading is the type safety property that prevents type spoofing. As explained by Liang and Bracha[26], by capturing events at runtime (new loaders, new class) and maintaining constraints on the multiple loaders and their delegation hierarchy, authors can avoid confusion when loading a spoofed class. This behavior is now implemented in modern Java virtual machines. Later Tazawa and Hagiya[42] proposed a formalization of the Java Virtual Machine supporting dynamic class loading in order to ensure type safety. Those works ensure strong safety for the Java Virtual Machine, in particular when linking new classes at runtime. Although Android has a similar mechanism, the implementation is not shared with the JVM of Oracle. Additionally, in this paper, we do not focus on spoofing classes at runtime, but on confusion that occurs when using a static analyzer used by a reverser that tries to understand the code loading process offline.

Contributions about Android class loading focus on using the capabilities of class loading to extend Android features or to prevent reverse engineering of Android applications. For instance, Zhou *et al.*[49] extend the class loading mechanism of Android to support regular Java bytecode and Kritz and Maly[21] propose a new class loader to automatically load modules of an application without user interactions.

Regarding reverse engineering, class loading mechanisms are frequently used by packers for hiding all or parts of the code of an application[9]. The problem to be solved consists in locating secondary `.dex` files that can be unciphered just before being loaded. Dynamic hook mechanisms should be used to intercept the bytecode at load time. These techniques can be of some help for the reverser, but they require to instrument the source code of AOSP or the application itself. The engineering cost is high and anti-debugging techniques can slow down the process. Thus, a reverser always starts by studying statically an application using static analysis tools[24], and will eventually go to dynamic analysis[10] if further costly extra analysis is needed (for example, if they spot the use of a custom class loader). In the first phase of an analysis where the used methods are static, the reverser can have the feeling that what he sees in the bytecode is what is loaded at runtime. Our goal is to show that tools mentioned in the literature[24] can suffer from attacks exploiting confusion inside regular class loading mechanisms of Android.

Hidden APIs Li *et al.* did an empirical study of the usage and evolution of hidden APIs[25]. They found that hidden APIs are added and removed in every release of Android, and that they are used both by benign and malicious applications. More recently, He *et al.* [17] did a systematic study of hidden service API related to security. They studied how the hidden API can be used to bypass Android security restrictions and found that although Google countermeasures are effective, they need to be implemented inside the system services and not the hidden API due

to the lack of in-app privilege isolation: the framework code is in the same process as the user code, meaning any restriction in the framework can be bypassed by the user.

Static analysis tools Static analysis tools are used to perform operations on an APK file, for example extracting its bytecode or information from the Manifest file. Because of the complexity of Android, few tools have followed all the evolutions of the file format and are robust enough to analyze all applications without crashing[32]. The tools can share the backend used to manipulate the code. For example, Apktool is often called in a subprocess to extracte the bytecode. Another example is Soot[2], a Java framework that allows to manipulate the bytecode from an object representation of instructions. This framework enables advanced features such as inserting or removing bytecode instructions but can require a lot of memory and time to perform its operations. The most known tool built on top of Soot is FlowDroid[3], which enables to compute information flows statically into the code.

Because these tools are used by reversers, we will evaluate the accuracy of the provided results in the case of an application developer exploits the possible confusions that brings the class loading mechanisms of Android.

5.3 Analyzing the class loading process

For building obfuscation techniques based on the confusion of tools with class loaders, we manually studied the code of Android that handles class loading. In this section, we report the inner workings of ART and we focus on the specificities of class loading that can bring confusion. Because the class loading implementation has evolved over time during the multiple iterations of the Android operating system, we mainly describe the behavior of ART from Android version 14 (SDK 34).

5.3.1 Class loaders

When ART needs to access a class, it queries a `ClassLoader` to retrieve its implementation. Each class has a reference to the `ClassLoader` that loaded it, and this class loader is the one that will be used to load supplementary classes used by the original class. For example in Listing 2, when calling `A.f()`, the ART will load `B` with the class loader that was used to load `A`.

```
class A {
    public static void f() {
        B b = new B();
        b.do_something();
}}
```

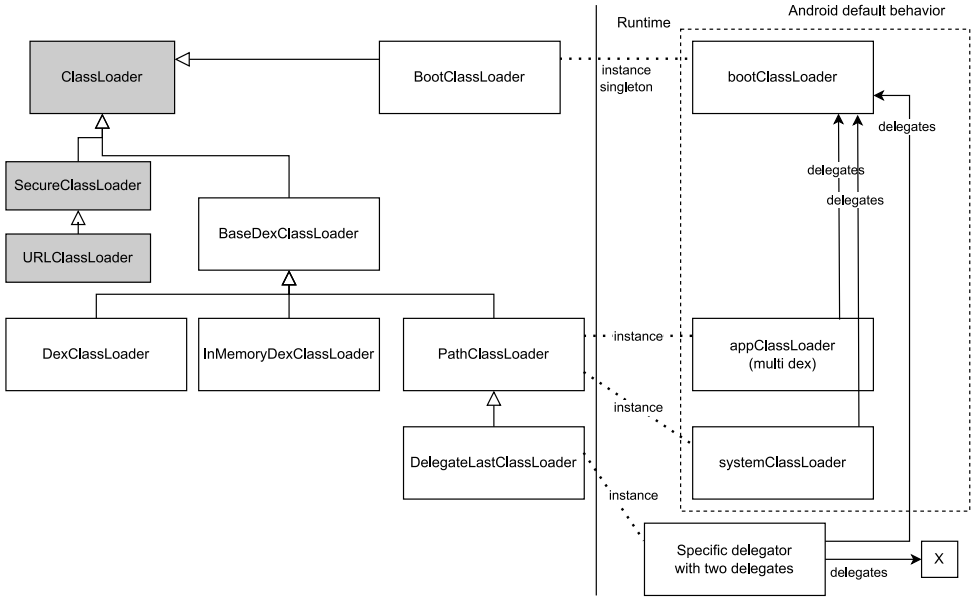
Listing 2: Class instantiation

This behavior has been inherited from Java and most of the core classes regarding class loaders have been kept in Android. Nevertheless, the Android implementation has slight differences and new class loaders have been added. For example, the java class loader `URLClassLoader` is still

present in Android, but contrary to the official documentation, most of its methods have been removed or replaced by a stub that just raises an exception. Moreover, rather than using the Java class loaders `SecureClassLoader` or `URLClassLoader`, Android has several new class loaders that inherit from `ClassLoader` and override the appropriate methods.

The left part of Figure 14 shows the different class loaders specific to Android in white and the stubs of the original Java class loaders in grey. The main difference between the original Java class loaders and the ones used by Android is that they do not support the Java bytecode format. Instead, the Android-specific class loaders load their classes from (many) different file formats specific to Android. Usually, when used by a programmer, the classes are loaded from memory or from a file using the DEX format (`.dex`). When used directly by ART, the classes are usually stored in an application file (`.apk`) or in an optimized format (`OAR/ODEX`).

TODO 19 ▶ *Alt text for cl-class_loading_classes* ◀



gray – Java-based, white – Android-based
Figure 14: The class loading hierarchy of Android

5.3.2 Delegation

The order in which classes are loaded at runtime requires special attention. All the specific Android class loaders (`DexClassLoader`, `InMemoryClassLoader`, etc.) have the same behavior (except `DelegateLastClassLoader`) but they handle specificities for the input format. Each class loader has a delegate class loader, represented in the right part of Figure 14 by black plain arrows for an instance of `PathClassLoader` and an instance of `DelegateLastClassLoader` (the other class

loaders also have this delegate). This delegate is a concept specific to class loaders and has nothing to do with class inheritance. By default, class loaders will delegate to the singleton class `BootClassLoader`, except if a specific class loader is provided when instantiating the new class loader. When a class loader needs to load a class, except for `DelegateLastClassLoader`, it will first ask the delegate, i.e. `BootClassLoader`, and if the delegate does not find the class, the class loader will try to load the class on its own. This behavior implements a priority and avoids re-defining by error a core class of the system, for example redefining `java.lang.String` that would be loaded by a child class loader instead of its delegates. `DelegateLastClassLoader` behaves slightly differently: it will first delegate to `BootClassLoader` then, it will check its files and finally, it will delegate to its actual delegate (given when instantiating the `DelegateLastClassLoader`). This behavior is useful for overriding specific classes of a class loader while keeping the other classes. A normal class loader would prioritize the classes of its delegate over its own.

```
def get_mutli_dex_classses_dex_name(index: int):
    if index == 0:
        return "classes.dex"
    else:
        return f"classes{index+1}.dex"

def load_class(class_name: str):
    if is_platform_class(class_name):
        return load_from_boot_class_loader(class_name)
    else:
        index = 0
        dex_file = get_mutli_dex_classses_dex_name(index)
        while file_exists_in_apk(dex_file) and \
            not class_found_in_dex_file(class_name, dex_file):
            index += 1
        if file_exists_in_apk(dex_file):
            return load_from_file(dex_file, class_name)
        else:
            raise ClassNotFoundError()
```

Listing 3: Default Class Loading Algorithm for Android Applications

At runtime, Android instantiates for each application three instances of class loaders described previously: `bootClassLoader`, the unique instance of `BootClassLoader`, and two instances of `PathClassLoader`: `systemClassLoader` and `appClassLoader`. `bootClassLoader` is responsible for loading Android **platform classes**. It is the direct delegate of the two other class loaders instantiated by Android. `appClassLoader` points to the application `.apk` file, and is used to load the classes inside the application `systemClassLoader` is a `PathClassLoader` pointing to `'.'`, the working directory of the application, which is `'/'` by default. The documentation of `ClassLoader.getSystemClassLoader` reports that this class loader is the default context

class loader for the main application thread. In reality, the platform classes are loaded by `bootClassLoader` and the classes from the application are loaded from `appClassLoader`. `systemClassLoader` is never used.

In addition to the class loaders instantiated by ART when starting an application, the developer of an application can use class loaders explicitly by calling to ones from the Android SDK, or by recoding custom class loaders that inherit from the `ClassLoader` class. At this point, modeling accurately the complete class loading algorithm becomes impossible: the developer can program any algorithm of their choice. For this reason, this case is excluded from this paper and we focus on the default behavior where the context class loader is the one pointing to the `.apk` file and where its delegate is `BootClassLoader`. With such a hypothesis, the delegation process can be modeled by the pseudo-code of method `load_class` given in .

In addition, it is important to distinguish the two types of platform classes handled by `BootClassLoader` and that both have priority over classes from the application at runtime:

- the ones available in the **Android SDK** (normally visible in the documentation);
- the ones that are internal and that should not be used by the developer. We call them **hidden classes**[17, 25] (not documented).

As a preliminary conclusion, we observe that a priority exists in the class loading mechanism and that an attacker could use it to prioritize an implementation over another one. This could mislead the reverser if they use the one that has the lowest priority. To determine if a class is impacted by the priority given to `BootClassLoader`, we need to obtain the list of classes that are part of Android *i.e.*, the platform classes. We discuss in the next section how to obtain these classes from the emulator.

5.3.3 Determining platform classes

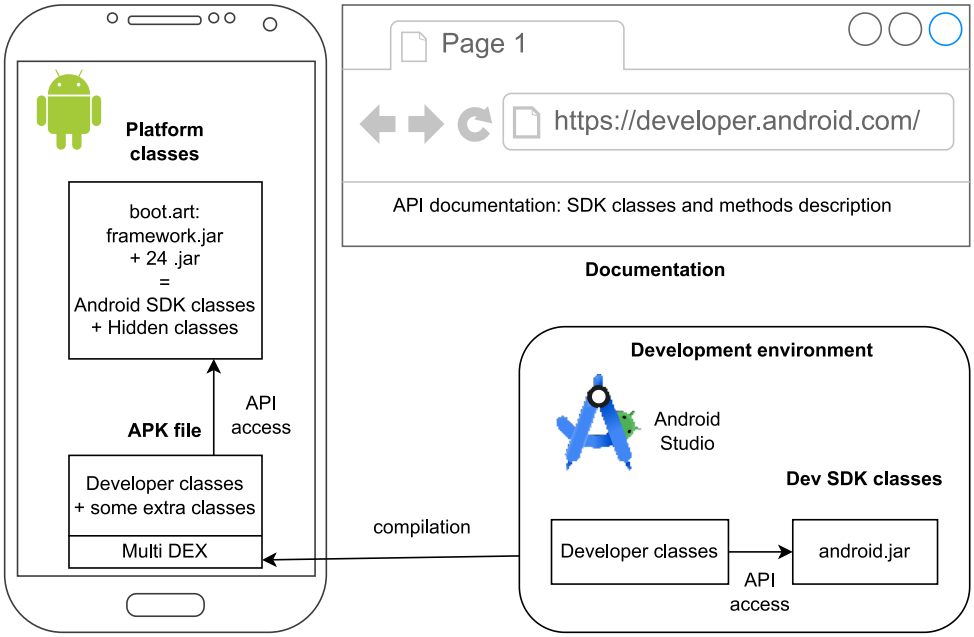


Figure 15: Location of SDK classes during development and at runtime

Figure 15 shows how classes of Android are used in the development environment and at runtime. In the development environment, Android Studio uses `android.jar` and the specific classes written by the developer. After compilation, only the classes of the developer, and eventually extra classes computed by Android Studio are zipped in the APK file, using the multi-dex format. At runtime, the application uses `BootClassLoader` to load the platform classes from Android. Until our work, previous works[17, 25] considered both Android SDK and hidden classes to be in the file `/system/framework/framework.jar` found in the phone itself, but we found that the classes loaded by `bootClassLoader` are not all present in `framework.jar`. For example, He *et al.* [17] counted 495 thousand APIs (fields and methods) in Android 12, based on Google documentation on restriction for non SDK interfaces[4]. However, when looking at the content of `framework.jar`, we only found 333 thousand APIs. Indeed, classes such as `com.android.okhttp.OkHttpClient` are loaded by `bootClassLoader`, listed by Google, but not in `framework.jar`.

For optimization purposes, classes are now loaded from `boot.art`. This file is used to speed up the start-up time of applications: it stores a dump of the C++ objects representing the **platform classes** (Android SDK and hidden classes) so that they do not need to be generated each time an application starts. Unfortunately, this format is not documented and not retro-compatible

4. https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

between Android versions and is thus difficult to parse. An easier solution to investigate the platform classes is to look at the `BOOTCLASSPATH` environment variable in an emulator. This variable is used to load the classes without the `boot.art` optimization. We found 25 `.jar` files, including `framework.jar`, in the `BOOTCLASSPATH` of the standard emulator for Android 12 (SDK 32), 31 for Android 13 (SDK 33), and 35 for Android 14 (SDK 35), containing respectively a total of 499 837, 539 236 and 605 098 API methods and fields. Table 5) summarizes the discrepancies we found between Google's list and the platform classes we found in Android emulators. Note also that some methods may also be found *only* in the documentation. Our manual investigations suggest that the documentation is not well synchronized with the evolution of the platform classes and that Google has almost solved this issue in API 34.

SDK version	**Number of API methods**			
	Documented	In emulator	Only documented	Only in emulator
32	495 713	499 837	1060	5184
33	537 427	539 236	1258	3067
34	605 106	605 098	26	18

Table 5: Comparison for API methods between documentation and emulators

We conclude that it can be dangerous to trust the documentation and that gathering information from the emulator or phone is the only reliable source. Gathering the precise list of classes and the associated bytecode is not a trivial task.

5.3.4 Multiple DEX files

For the application class files, Android uses its specific format called DEX: all the classes of an application are loaded from the file `classes.dex`. With the increasing complexity of Android applications, the need arrised to load more methods than the DEX format could support in one `.dex` file. To solve this problem, Android started storing classes in multiple files named `classesX.dex` as illustrated by the Listing 4 that generates the filenames read by class loaders. Android starts loading the file `GetMultiDexClassesDexName(0)` (`classes.dex`), then `GetMultiDexClassesDexName(1)` (`classes2.dex`), and continues until finding a value `n` for which `GetMultiDexClassesDexName(n)` does not exist. Even if Android emits a warning message when it finds more than 100 `.dex` files, it will still load any number of `.dex` files that way. This change had the unintended consequence of permitting two classes with the same name but different implementations to be stored in the same `.apk` file using two `.dex` files.

Android explicitly performs checks that prevent several classes from using the same name inside a `.dex` file. However, this check does not apply to multiple `.dex` files in the same `.apk` file, and

a `.dex` can contain a class with a name already used by another class in another `.dex` file of the application. Of course, such a situation should not happen when multiple `.dex` files have been generated by properly Android Studio. Nevertheless, for an attacker controlling the process, this issue raises the question of which class is selected when several classes sharing the same name are present in `.apk` files.

We found that Android loads the class whose implementation is found first when looking in the order of multiple `dexfiles`, as generated by the method `GetMultiDexClassesDexName`. We will show later in Section 5.4.2 that this choice is not the most intuitive and can lead to fool analysis tools when reversing an application. As a conclusion, we model both the multi-dex and delegation behaviors in the pseudo-code of Listing 3.

```
++
std::string DexFileLoader::GetMultiDexClassesDexName(size_t index) {
  return (index == 0) ?
    "classes.dex" :
    StringPrintf("classes%zu.dex", index + 1);
}
```

Listing 4: The method generating the .dex filenames from the AOSP

5.4 Obfuscation Techniques

In this section, we present new obfuscation techniques that take advantage of the complexity of the class loading process. Then, in order to evaluate their efficiency, we reviewed some common Android reverse analysis tools to see how they behave when collisions occur between classes of the APK or between a class of the APK and classes of Android (Android SDK or hidden classes). We call this collision "**class shadowing**", because the attacker version of the class shadows the one that will be used at runtime. To evaluate if such shadow attacks are working, we handcrafted three applications implementing shadowing techniques to test their impact on static analysis tools. Then, we manually inspected the output of the tools in order to check its consistency with what Android is really doing at runtime. For example, for Apktool, we look at the output disassembled code, and for Flowdroid[3], we check that a flow between `Taint.source()` and `Taint.sink()` is correctly computed.

5.4.1 Obfuscation Techniques

From the results presented in Section 5.3, three approaches can be designed to hide the behavior of an application.

Self shadow*: shadowing a class with another from APK* This method consists in hiding the implementation of a class with another one by exploiting the possible collision of class names, as described in Section 5.3.4 with multiple `.dex` files. If reversers or tools ignore the priority order of a multi-dex file, they can take into account the wrong version of a class.

SDK shadow*: shadowing a SDK class* This method consists in presenting to the reverser a fake implementation of a class of the SDK. This class is embedded in the APK file and has the same name as the one of the SDK. Because `BootClassLoader` will give priority to the Android SDK at runtime, the reverser or tool should ignore any version of a class that is contained in the APK. The only constraint when shadowing an SDK class is that the shadowing implementation must respect the signature of real classes. Note that, by introducing a custom class loader, the attacker could inverse the priority, but this case is out of the scope of this paper.

Hidden shadow*: shadowing an hidden class* This method is similar to the previous one, except the class that is shadowed is a hidden class. Because ART will give priority to the internal version of the class, the version provided in the APK file will be ignored. Such shadow attacks are more difficult to detect by a reverser, that may not know the existence of this specific hidden class in Android.

5.4.2 Impact on static analysis tools

```
public class Main {
    public static void main(Activity ac) {
        String personal_data = Taint.source();
  String obfuscated_personal_data = Obfuscation.hide_flow(personal_data);
  Taint.sink(ac, obfuscated_personal_data);
    }
}
public class Obfuscation { // customized for each obfuscation technique
  public static String hide_flow(String personal_data) { ... }
```

Listing 5: Main body of test apps

We selected tools that are commonly used to unpack and reverse Android applications: Jadx[5], a decompiler for Android applications, Apktool[6], a disassembler/repackager of applications, Androguard[7], one of the oldest Python package for manipulating Android applications, and Flowdroid[3] that performs taint flow analysis.

For evaluating the tools, we designed a single application that we can customize for different tests. Listing 5 shows the main body implementing:

- a possible flow to evaluate FlowDroid: a flow from a method `Taint.source()` to a method `Taint.sink(Activity, String)` through a method `Obfuscation.hide_flow(String)`;
- a possible use of a SDK or hidden class inside the class `Obfuscation` to evaluate platform classes shadowing for other tools.

5. https://github.com/skylot/jadx
6. https://apktool.org/
7. https://github.com/androguard/androguard

The first application we released is a control application that does not do anything special. It will be used for checking the expecting result of tools. The second implements self shadowing: the class `Obfuscation` is duplicated: one is the same as the in the control app (`Obfuscation.hide_flow(String)` returns its arguments), and the other version returns a constant string. These two versions are embedded in several DEX of a multi-dex application. The third application tests SDK shadowing and needs an existing class of the SDK. We used `Pair` that we try to shadow. We put data in a `Pair` and reread the data from the `Pair`. The colliding `Pair` discards the data and returns null. The last application tests for Hidden API shadowing. Like for the third one, we similarly store data in `com.android.okhttp.Request` and then retrieve it. Again, the shadowing implementation discards the data.

We found that these static analysis tools do not consider the class loading mechanism, either because the tools only look at the content of the application file (*e.g.*, a disassembler) or because they consider class loading to be a dynamic feature and thus out of their scope. In Table 6, we report on the types of shadowing that can be tricked each tool. A plain circle is a shadow attack that leads to a wrong result. A white circle indicates a tool emitting warnings or that eventually displays the two versions of the class. A cross is a tool not impacted by a shadow attack. We explain in more detail in the following the results for each considered tool.

Tool	Version	Shadow Attack		
		Self	SDK	Hidden
Jadx	1.5.0	○	●	●
Apktool	2.9.3	○	●	●
Androguard	4.1.2	○	●	●
Flowdroid	2.13.0	●	×	●

●: working
○: works but producing warning or can be seen by the reverser
×: not working

Table 6: Working attacks against static analysis tools

5.4.2.1 Jadx

Jadx is a reverse engineering tool that regenerates the Java source code of an application. It processes all the classes present in the application, but only save/display one class by name, even if two versions are present in multiple `.dex` files. Nevertheless, when multiple classes with the same name are found, Jadx reports it in a comment added to the generated Java source code. This warning stipulates that a possible collision exists and lists the files that contain

the different versions of the class. Unfortunately, after reviewing the code of Jadx, we believe that the selection of the displayed class is an undefined behavior. At least for the version 1.5.0 that we tested, we found that Jadx selects the wrong implementation when a class with the same name is present. For example in `classes2.dex` and `classes3.dex`. We report it with a "o" because warnings are issued.

Shadowing Android SDK and hidden classes is possible in Jadx: there is only one implementation of the class in the application and Jadx does not have a list of the internal classes of Android: no warning is issued to the reverser that the displayed class is not the one used by Android.

5.4.2.2 Apktool

Apktool generates Smali files, an assembler language for DEX bytecode. Apktool will store the disassembled classes in a folder that matches the `.dex` file that stores the bytecode. This means that when shadowing a class with two versions in two `.dex` files, the shadow implementations will be disassembled into two directories. No indication is displayed that a collision is possible. It is up to the reverser to have a chance to open the good one.

Similarly to Jadx, using an Android SDK or hidden class will not be detected by the tool that will unpack the fake shadow version.

5.4.2.3 Androguard

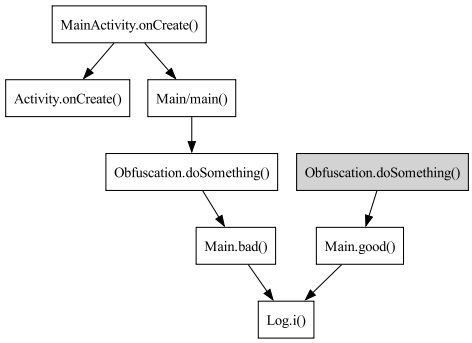
Androguard has different usages, with different levels of analysis. The documentation highlights the analysis commands that compute three types of objects: an APK object, a list of DEX objects, and an Analysis object. The APK and the list of `.dex` files are a one-to-one representation of the content of an application, and have the same issues that we discussed with Apktool: they provide the different versions of a shadow class contained in multiple `.dex` files.

The Analysis object is used to compute a method call graph and we found that this algorithm may choose the wrong version of a shadowed class when using the cross references that are computed. This leads to an invalid call graph as shown in Subfigure 18: the two methods `doSomething()` are represented in the graph, but the one linked to `main()` on the graph is the one calling the method `good()` when in fact the method `bad()` is called when running the application.

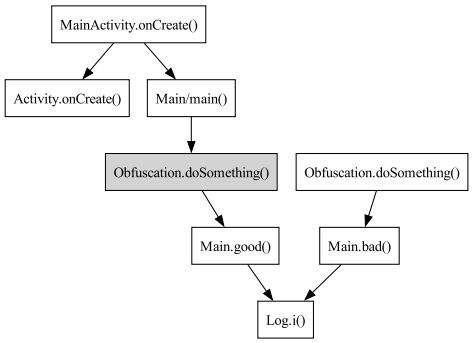
Androguard has a method `.is_external()` to detect if the implementation of a class is not provided inside the application and a method `.is_android_api()` to detect if the class is part of the Android API. Regrettably, the documentation of `.is_android_api()` explains that the method is still experimental and just checks a few package names. This means that although those methods are useful, the only indication of the use of an Android SDK or hidden classes is the fact that the class is not in the APK file. Because of that, like for Apktool and Jadx,

Androguard has no way to warn the reverser that the shadow of an Android SDK or hidden classes is not the class used when running the application.

TODO 20 ▶ *alt text androguard_call_graph* ◀



Subfigure 17: Expected Call Graph



Subfigure 18: Call Graph Computed by Androguard

Figure 16: Call Graphs of an application calling `Main.bad()` from a shadowed `Obfuscation` class.

5.4.2.4 Flowdroid

Flowdroid[3] is used to detect if an application can leak sensitive information. To do so, the analyst provides a list of source and sink methods. The return value of a method marked as source is considered sensitive and the argument of a method marked as sink is considered to be leaked. By analyzing the bytecode of an application, Flowdroid can detect if data emitted by source methods can be exfiltrated by a sink method. Flowdroid is built on top of the Soot[2] framework that handles, among other things, the class selection process.

We found that when selecting the classes implementation in a multi-dex APK, Soot uses an algorithm close to what ART is performing: Soot sorts the `.dex` bytecode file with a specified `prioritizer` (a comparison function that defines an order for `.dex` files) and selects the first implementation found when iterating over the sorted files. Unfortunately, the `prioritizer` used by Soot is not exactly the same as the one used by the ART. The Soot `prioritizer` will give priority to `classes.dex` and then give priority to files whose name starts with `classes` over other files and finally will use the alphabetical order. This order is good enough for application with a small number of `.dex` files generated by Android Studio, but because it uses the alphabetical order and does not check the exact format used by Android, a malicious developer could hide the implementation of a class in `classes2.dex` by putting a false implementation in `classes0.dex`, `classes1.dex` or `classes12.dex`.

In addition to self shadowing, Flowdroid is sensitive to the use of platform classes, as it needs the bytecode of those classes to be able to track data flows. This is solved for SDK classes by providing `android.jar` to Flowdroid. Flowdroid gives priority to the classes from the SDK over the classes implemented in the application, thus defeating SDK shadow attacks. Unfortunately, `android.jar` only contains classes from the Android SDK, meaning that using hidden classes breaks the flow tracking. Solving this issue would require finding the bytecode of all the platform classes of the Android version targeted and as we said previously it requires extracting this information from the emulator.

We have seen that tools can be impacted by shadow attacks. In the next section, we will investigate if these attacks are used in the wild.

5.5 Shadow attacks in the wild

In this section, we evaluate in the wild if applications that can be found in the Play store or other markets use one of the shadow techniques. Our goal is to explore the usage of shadow techniques in real applications. Because we want to include malicious applications (in case such techniques would be used to hide malicious code), we selected 50 000 applications randomly from AndroZoo[1] that appeared in 2023. Malicious applications are spot in our dataset by using a threshold of 3 over the number of antivirus reporting an application as a malware. Some few applications over the total cannot be retrieved or parsed leading to a final dataset of 49 975 applications. We automatically disassembled the applications to obtain the list of included classes. Then, we check if any shadow attack occurs in the APK itself or with platform classes of SDK 34.

5.5.1 Results

TODO 21 ▶ *cl-shadow* ◀

	Number of apps			Average				Identical Code
		%	% malware	Shadow classes	Median	Target SDK	Min SDK	
For all applications of the dataset								
Self	49 975	100.0%	0.53%	2.1	0	32.1	21.7	74.8%
Sdk	49 975	100.0%	0.53%	6.5	0	32.1	21.7	8.04%
Hidden	49 975	100.0%	0.53%	0.5	0	32.1	21.7	17.42%
Total	49 975	100.0%	0.53%	9	0	32.1	21.7	23.76%
For applications with at least 1 shadow case								
Self	234	0.47%	5.98%	438.1	18	31.4	22.4	74.8%
Sdk	11 755	23.52%	0.38%	27.6	5	32.4	22	8.04%
Hidden	1556	3.11%	0.71%	16.1	1	32.1	22.2	17.42%
Total	12 301	24.61%	0.42%	36.7	6	32.4	22	23.76%

Table 7: Shadow classes compared to SDK 34 for a dataset of 49 975 applications

We report in the upper part of Table 7 the statistics about the whole dataset and the three shadow attacks: "self" when a class shadows another one in the APK, "SDK" when a class of the SDK shadows one of the APK, and "Hidden" when a hidden class of Android shadows one of the APK. We observe that, on average, a few classes are shadowed by another class. Note that the median value is 0 meaning that few apps shadow a lot of classes, but the majority of apps do not shadow anything. The number of applications shadowing a hidden API is low, which is an expected result as these classes should not be known by the developer. We observe a consequent number of applications, 23.52%, of applications that perform SDK shadowing. It can be explained by the fact that some classes that newly appear are embedded in the APK for end users that have old versions of Android: it is suggested by the average value of Min SDK which is 21.7 for the whole dataset: on average, an application can be run inside a smartphone with API 21, which would require to embed all new classes from 22 to 34. This hypothesis about missing classes is further investigated later in this section.

In the bottom part of Table 7, we give the same statistics but we excluded applications that do not perform any shadowing. For those pairs of shadow classes, we disassembled them using Apktool to perform a comparison using instructions represented in the Smali language. For self-shadow, we compare the pair. For the shadowing of the SDK or Hidden class, we compare the code found in the APK with implementations found in the emulator and `android.jar` of SDK 32, 33, and 34.

Self-shadowing We observe a low number of applications doing self-shadow attacks. For each class that is shadowed, we compared its bytecode with the shadowed one. We observe that 74.8% are identical which suggests that the compilation process embeds the same class multiple times but makes variations in headers or metadata values. We investigate later in Section 5.5.2 the case of malicious applications.

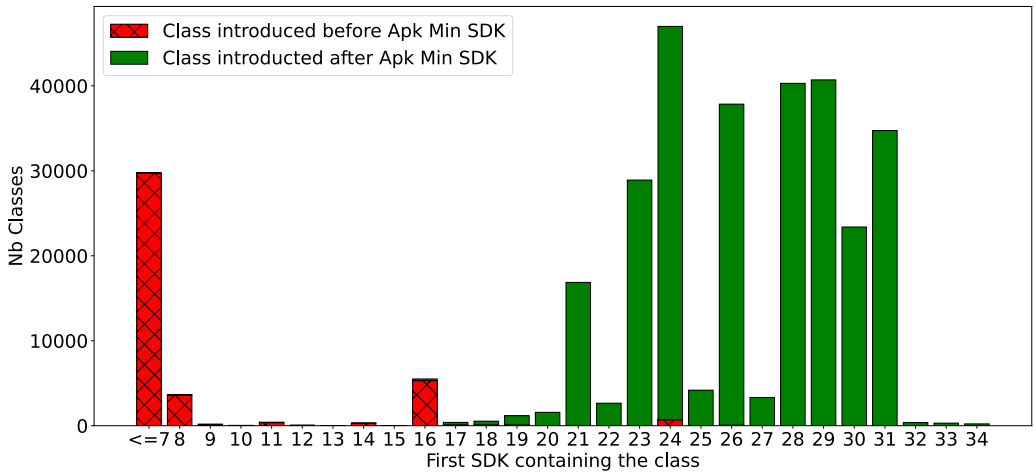


Figure 19: Redefined SDK classes, sorted by the first SDK they appeared in.

SDK shadowing For the shadowing of SDK classes, we observe a low ratio of identical classes. This result could lead to the wrong conclusion that developers embed malicious versions of the SDK classes, but our manual investigation shows that the difference is slight and probably due to compiler optimization. To go further in the investigation, in Figure 19 we represent these redefined classes with the following rules:

- The class is classified on the X abscissa in the figure according to the SDK it first appeared in.
- The class is counted as "green" (solid) if it first appeared in the SDK **after** the APK min SDK (retro compatibility purpose).
- The class is counted as "red" (hatched) if it first appeared in the SDK **before** the APK min SDK (which is useless for the application as the SDK version is always available).

We observe that the majority of classes are legitimate retro-compatibility additions of classes, especially after SDK 21 (which is the average min SDK, cf. Table 7). Abnormal cases are observed for classes that appeared in API versions 7 and before, 8, and 16. Table 8 reports the top ten classes that shadow the SDK for the three mentioned versions. For SDK before 7, it mainly concerns HTTP classes: for example, the class `HttpParams` is an interface, containing limited bytecode that mostly matches the class already present on the emulator (98.03% of shadowed classes are identical). `HttpConnectionParams` on the other hand differs from the platform class and we observe only 4.99% of identical classes. Manual inspection of some applications revealed that the two main reasons are:

- instead of checking if the methods attributes are null inline like Android does, applications use the method `org.apache.http.util.Args.notNull()`. According to comments in the source

code of Android[8], the class was forked in 2007 from Apache 'httpcomponents' project. Looking at the history of the project, the use of `Args.notNull()` was introduced in 2012[9]. This shows that applications are embedding code from more recent version of this library without realizing their version will not be the used one.

- very small changes that we found can be attributed to the compilation process (e.g. swapping registers: `v0` is used instead of `v1` and `v1` instead of `v0`), but even if we consider them different, they are very similar.

The remaining 4.99% of classes that are identical to the Android version are classes where the body of the methods is replaced by stubs that throw `RuntimeException("Stub!")`. This code corresponds to what we found in android.jar but not the code we found in the emulator, which is surprising. Nevertheless, we decided to count them as identical, because `android.jar` is the official jar file for developer, and stubs are replaced in the emulator: it is intended by Google developers.

Other results of Table 8 can be similarly discussed: either they are identical with a high ratio, or they are different because of small variations. When substantial differences appear it is mainly because different versions of the same library have been used or an SDK class is embedded for retro-compatibility.

TODO 22 ▶ ***cl-topsdk*** ◀

8. https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/core/java/org/apache/http/params/HttpConnectionParams.java;drc=3bdd327f8532a79b83f575cc62e8eb09a1f93f3d?
9. https://github.com/apache/httpcomponents-core/commit/9104a92ea79e338d876b1b60f5cd2b243ba7069f?

Class	Occurrences	Identical ratio
redefined for SDK ≤ 7		
`Lorg/apache/http/params/HttpParams;`	1318	98.03%
`Lorg/apache/http/params/HttpConnectionParams;`	1202	4.99%
`Lorg/apache/http/conn/ConnectTimeoutException;`	1200	35.0%
`Lorg/apache/http/params/CoreConnectionPNames;`	1190	99.92%
`Lorg/xmlpull/v1/XmlPullParser;`	1111	52.57%
`Lorg/apache/http/conn/scheme/SocketFactory;`	1074	87.52%
`Lorg/apache/http/conn/scheme/HostNameResolver;`	1072	87.59%
`Lorg/apache/http/conn/scheme/LayeredSocketFactory;`	963	89.41%
`Lorg/json/JSONException;`	945	0.0%
`Lorg/apache/http/conn/ssl/X509HostnameVerifier;`	886	0.79%
redefined for SDK $= 8$		
`Ljavax/xml/namespace/QName;`	297	0.0%
`Ljavax/xml/namespace/NamespaceContext;`	226	98.23%
`Landroid/net/http/SslError;`	221	31.67%
`Lorg/w3c/dom/UserDataHandler;`	82	92.68%
`Ljavax/xml/transform/TransformerConfigurationException;`	73	69.86%
`Ljavax/xml/transform/TransformerException;`	73	0.0%
`Lorg/w3c/dom/ls/LSException;`	61	63.93%
`Lorg/w3c/dom/TypeInfo;`	54	88.89%
`Lorg/w3c/dom/DOMConfiguration;`	54	46.3%
`Ljavax/xml/transform/TransformerFactoryConfigurationError;`	52	0.0%
redefined for SDK $= 16$		
`Landroid/annotation/SuppressLint;`	2634	98.48%
`Landroid/annotation/TargetApi;`	2634	98.48%
`Landroid/media/MediaCodec$CryptoException;`	11	18.18%
`Landroid/media/MediaCryptoException;`	10	20.0%
`Landroid/view/accessibility/AccessibilityNodeProvider;`	9	0.0%
`Landroid/view/ActionProvider$VisibilityListener;`	8	12.5%
`Landroid/app/Notification$BigTextStyle;`	7	0.0%
`Landroid/app/Notification$Style;`	7	0.0%
`Landroid/util/LongSparseArray;`	7	0.0%
`Landroid/media/MediaPlayer$TrackInfo;`	7	0.0%

Table 8: Shadow classes compared to SDK 34 for a dataset of 49 975 applications

Hidden shadowing For applications redefining hidden classes, on average, 16.1 classes are redefined (cf bottom part of Table 7). The top 3 packages whose code actually differs from the ones found in Android are `java.util.stream`, `org.ccil.cowan.tagsoup` and `org.json`:

- stream: when looking in more detail, we found that `java.util.stream` was only redefined by 6 applications, but the large number of classes redefined artificially puts the package at the top of the list.
- tagsoup: `TagSoup` is a library for parsing HTML.
- json: there is only one hidden class in `org.json`, redefined by 821 applications: `JSONObject$1`. `org.json` is a package in Android SDK, not a hidden one. However, `JSONObject$1` is an

anonymous class not provided by `android.jar` because its class `JSONObject` is an empty stub, and thus, does not use `JSONObject$1`. Thus, this class falls in the category of hidden platform classes.

All these hidden shadow classes are libraries included by the developers who probably did not know that they were already embedded in Android.

5.5.2 Shadowing in malware applications

```
public class Reflection {
  private static final int ERROR_SET_APPLICATION_FAILED = -20;
  private static final String TAG = "Reflection";
  // ...

  static {
    try {
      Method declaredMethod = Class.class.getDeclaredMethod("forName", String.class);
      Method declaredMethod2 = Class.class.getDeclaredMethod("getDeclaredMethod",
String.class, Class[].class);
      Class cls = (Class) declaredMethod.invoke(null, "dalvik.system.VMRuntime");
      Method method = (Method) declaredMethod2.invoke(cls, "getRuntime", null);
      setHiddenApiExemptions = (Method) declaredMethod2.invoke(cls,
"setHiddenApiExemptions", new Class[]{String[].class});
      sVmRuntime = method.invoke(null, new Object[0]);
    } catch (Throwable th) { Log.e(TAG, "reflect bootstrap failed:", th); }
    System.loadLibrary("free-reflection");
    // ...
  }
  // ...
}
```

Listing 6: Implementation of Reflection found un classes11.dex (shadows Listing 7)

```
public class Reflection {
  private static final String DEX = "ZGV4CjAzNQCl4EprGS2pXI/
v3OwlBrlfRnX5rmkKVdN0CwAAcA ... AoAAA==";
  private static final String TAG = "Reflection";

  private static native int unsealNative(int i);

  public static int unseal(Context context) {
    return (Build.VERSION.SDK_INT < 28 || BootstrapClass.exemptAll() ||
unsealByDexFile(context)) ? 0 : -1;
  }

  private static boolean unsealByDexFile(Context context) {
    // Decode DEX from base64 and load it as bytecode.
    // ...
  }
  // ...
}
```

Listing 7: Implementation of Reflection executed by ART (shadowed by Listing 6

The last column of Table 7 shows the proportion of applications considered as malware because we arbitrarily fixed a threshold of 3 positive detections from VirusTotal reports. For the whole dataset, we have 0.53% of applications considered as malware. We can see that an application that uses self-shadowing is 10 times more likely to be a malware, when the proportion of malware among application shadowing platform classes is the same as in the rest of the dataset. Thus, we manually reversed self-shadowing malware, and found that the self-shadowing does not look to be voluntary. The colliding classes are often the same implementation, occasionally with minor differences, like different versions of a library. Additionally, we noticed multiple times internal classes from `com.google.android.gms.ads` colliding with each other, but we believe that it is due to bad processing during the compilation of the application.

The most notable case we found was an application that still exists on the Google Play Store with the same package name[10]. This application contains a self-shadow class `me.weishu.reflection.Reflection` that can be found in github, in the repository `tiann/FreeReflection`[11]. This class is used to disable Android restrictions on hidden API. At first glance, we believed the shadowing to be done voluntarily for obfuscation purposes. The shadow class that would be seen by a reverser is given in Listing 6: it contains some Java bytecode performing reflection and loading a native library named "free-reflection" (the associated `.so` is missing). The shadowed class that is really executed is summarized in Listing 7. It contains a more obfuscated code: a `DEX` field storing base64 encoded DEX bytecode that is later used to

10. SHA256: `C46A65EA1A797119CCC03C579B61C94FE8161308A3B6A8F55718D6ADAD112546`
11. https://github.com/tiann/FreeReflection

load some new code. When looking at this new code stored in the field, we found that it does almost the same thing as the code in the shadow class. Thus, we believe that the developer has upgraded their obfuscation techniques, replacing a native library by inline base64 encoded bytecode. The shadow attack could be unintentional, but it strengthens the masking of the new implementation.

As a conclusion, we observed that:

- SDK shadowing is performed by 23.52% of applications but are unintentional: these classes are embedded for retro-compatibility purpose or because the developer added a library already present in Android;
- Hidden shadowing rarely occurs and is mainly due to the usage of libraries that Android already contains;
- Malware perform more self-shadowing than goodware applications, and we found a sample where self-shadowing would clearly mislead the reverser.

5.6 Threat to validity

During the analysis of the ART internals, we made the hypothesis that its different operating modes are equivalent: we analyzed the loading process for classes stored as non-optimized `.dex` format, and not for the pre-compiled `.oat`. It is a reasonable hypothesis to suppose that the two implementations have been produced from the same algorithm using two compilation workflows. Similarly, we assumed that the platform classes stored in `boot.art` are the same as the ones in `BOOTCLASSPATH`. We confirm empirically our hypothesis on an Android Emulator, but we may have missed some edge cases.

The comparison of Smali code can lead to underestimated values, for example, if the compilation process performs minor modifications such as instruction reordering. The ratios reported in this study for the comparison of code are thus a lower bound and would be higher with a more precise comparison. In addition, platform classes are stored differently in older versions of Android and could not be easily retrieved. For this reason, we did not compared the classes found in applications to their versions older than SDK 32 to avoid producing unreliable statistics for those versions.

5.7 Conclusion

This paper has presented three shadow attacks that allow malware developers to fool static analysis tools when reversing an Android application. By including multiple classes with the same name or by using the same name as a class of the Android SDK, the developer can mislead a reverser or impact the result of a flow analysis, such as the ones of Androguard or Flowdroid.

We explored if such shadow attacks are present in as dataset of 49 975 applications . We found that on average, 23.52% of applications are shadowing the SDK, mainly for retro-compatibility

purposes and library embedding. More suspiciously, 3.11% of applications are shadowing a hidden class, which could lead to unexpected execution as these classes can appear/disappear with the evolution of Android internals. Investigations for applications that defined classes multiple times suggest that the compilation process or the inclusion of different versions of the same library is the main explanation. Finally, when investigating malware samples, we found a specific sample containing a shadow attack that would hide a part of the critical code from a reverser studying the application.

Future work concerns the correctness of bytecode analysis. For now, we rely on the Smali representation of the bytecode but the compilation process makes this comparison difficult. We intend to better parse the bytecode to summarize it and be able to have a more reliable comparison method.

Chapter 6

Contribution n

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a Polyaeno, familiari suo, geometrica discere maluisset quam illum etiam ipsum dedocere. Sol Democrito magnus videtur, quippe homini erudito in geometriaque perfecto, huic pedalis fortasse; tantum enim esse omnino in nostris poetis aut inertissimae segnitiae est aut fastidii delicatissimi. Mihi quidem videtur, inermis ac nudus est. Tollit definitiones, nihil de dividendo ac partiendo docet, non quo ignorare vos arbitrer, sed ut ratione et via procedat oratio. Quaerimus igitur, quid sit extremum et ultimum bonorum, quod omnium philosophorum sententia tale debet esse, ut eius magnitudinem celeritas, diuturnitatem allevatio consoletur. Ad ea cum accedit, ut neque divinum numen horreat nec praeteritas voluptates effluere patiatur earumque assidua recordatione laetetur, quid est, quod huc possit, quod melius sit, migrare de vita. His rebus instructus semper est in voluptate esse aut in armatum hostem impetum fecisse aut in poetis evolvendis, ut ego et Triarius te hortatore facimus, consumeret, in quibus hoc primum est in quo admirer, cur in gravissimis rebus non delectet eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non inviti legant. Quis enim tam inimicus paene nomini

Romano est, qui Ennii Medeam aut Antiopam Pacuvii spernat aut reiciat, quod se isdem Euripidis fabulis delectari dicat, Latinas litteras oderit? Synephebos ego, inquit, potius Caecilii aut Andriam Terentii quam utramque Menandri legam? A quibus tantum dissentio, ut, cum Sophocles vel optime scripserit Electram, tamen male conversam Atilii mihi legendam putem, de quo Lucilius: 'ferreum scriptorem', verum, opinor, scriptorem tamen, ut legendus sit. Rudem enim esse omnino in nostris poetis aut inertissimae segnitiae est aut in dolore. Omnis autem privatione doloris putat Epicurus.

Chapter 7

Conclusion

TODO 23 ▶ *Conclude* ◀

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a Polyaeno, familiari suo, geometrica discere maluisset quam illum etiam ipsum dedocere. Sol Democrito magnus videtur, quippe homini erudito in geometriaque perfecto, huic pedalis fortasse; tantum enim esse omnino in nostris poetis aut inertissimae segnitiae est aut fastidii delicatissimi. Mihi quidem videtur, inermis ac nudus est. Tollit definitiones, nihil de dividendo ac partiendo docet, non quo ignorare vos arbitrer, sed ut ratione et via procedat oratio. Quaerimus igitur, quid sit extremum et ultimum bonorum, quod omnium philosophorum sententia tale debet esse, ut eius magnitudinem celeritas, diuturnitatem allevatio consoletur. Ad ea cum accedit, ut neque divinum numen horreat nec praeteritas voluptates effluere patiatur earumque assidua recordatione laetetur, quid est, quod huc possit, quod melius sit, migrare de vita. His rebus instructus semper est in voluptate esse aut in armatum hostem impetum fecisse aut in poetis evolvendis, ut ego et Triarius te hortatore facimus, consumeret, in quibus hoc primum

est in quo admirer, cur in gravissimis rebus non delectet eos sermo patrius, cum idem fabellas Latinas ad verbum e Graecis expressas non inviti legant. Quis enim tam inimicus paene nomini Romano est, qui Ennii Medeam aut Antiopam Pacuvii spernat aut reiciat, quod se isdem Euripidis fabulis delectari dicat, Latinas litteras oderit? Synephebos ego, inquit, potius Caecilii aut Andriam Terentii quam utramque Menandri legam? A quibus tantum dissentio, ut, cum Sophocles vel optime scripserit Electram, tamen male conversam Atilii mihi legendam putem, de quo Lucilius: 'ferreum scriptorem', verum, opinor, scriptorem tamen, ut legendus sit. Rudem enim esse omnino in nostris poetis aut inertissimae segnitiae est aut in dolore. Omnis autem privatione doloris putat Epicurus.

Bibliography

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *13th Working Conference on Mining Software Repositories (MSR)*, May 2016. 468–471.

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting Android and Java Applications as Easy as abc. In *Fourth International Conference on Runtime Verification*, September 2013. Springer Berlin Heidelberg, Rennes, France, 364–381. https://doi.org/10.1007/978-3-642-40787-1_26

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 05, 2014. ACM Press, Edinburgh, UK, 259–269. https://doi.org/10.1145/2666356.2594299

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013. ACM, 641–660. https://doi.org/10.1145/2509136.2509549

[5] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. *ACM SIGPLAN Notices* 48, 6 (June 2013), 275–286. https://doi.org/10.1145/2499370.2462186

[6] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, April 02, 2017. ACM, Abu Dhabi United Arab Emirates, 71–85. https://doi.org/10.1145/3052973.3053004

[7] Kwanghoon Choi and Byeong-Mo Chang. 2014. A Type and Effect System for Activation Flow of Components in Android Programs. *Information Processing Letters* 114, 11 (2014), 620–627. https://doi.org/10.1016/j.ipl.2014.05.011

[8] Anthony Desnos and Geoffroy Gueguen. 2011. Android: From Reversing to Decompilation. *Black Hat Abu Dhabi* (2011). Retrieved from https://media.blackhat.com/bh-ad-11/Desnos/bh-ad-11-DesnosGueguen-Andriod-Reversing_to_Decompilation_WP.pdf

[9] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and Xiaofeng Wang. 2018. Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation. In *24th Annual Network and Distributed System Security Symposium*, 2018.

[10] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys* 44, 2 (2012). https://doi.org/10.1145/2089125.2089126

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010. USENIX Association, Vancouver, BC, Canada, 393–407.

[12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, October 16, 2012. ACM, Raleigh North Carolina USA, 50–61. https://doi.org/10.1145/2382196.2382205

[13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, November 04, 2013. ACM, Berlin Germany, 45–54. https://doi.org/10.1145/2517312.2517315

[14] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Pasquale Stirparo. 2015. A Permission Verification Approach for Android Mobile Applications. *Computers & Security* 49, (March 2015), 192–205. https://doi.org/10.1016/j.cose.2014.10.005

[15] Li Gong. 1998. Secure Java class loading. *IEEE Internet Computing* 2, 6 (November 1998), 56–61. https://doi.org/10.1109/4236.735987

[16] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015. The Internet Society.

[17] Yi He, Yacong Gu, Purui Su, Kun Sun, Yajin Zhou, Zhi Wang, and Qi Li. 2023. A Systematic Study of Android Non-SDK (Hidden) Service API Security. *IEEE Transactions on Dependable and Secure Computing* 20, 2 (March 2023), 1609–1623. https://doi.org/10.1109/TDSC.2022.3160872

[18] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, March 18, 2013. Association for Computing Machinery, New York, NY, USA, 1844–1851. https://doi.org/10.1145/2480362.2480706

[19] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, October 19, 2012. ACM, Raleigh North Carolina USA, 3–14. https://doi.org/10.1145/2381934.2381938

[20] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, June 12, 2014. ACM, Edinburgh United Kingdom, 1–6. https://doi.org/10.1145/2614628.2614633

[21] Pavel Kriz and Filip Maly. 2015. Provisioning of application modules to Android devices. In *2015 25th International Conference Radioelektronika (RADIOELEKTRONIKA)*, April 2015. 423–426. https://doi.org/10.1109/RADIOELEK.2015.7129009

[22] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 280–291. https://doi.org/10.1109/ICSE.2015.48

[23] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *ICT Systems Security and Privacy Protection*, 2015. Springer International Publishing, Cham, 513–527. https://doi.org/10.1007/978-3-319-18467-8_34

[24] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* 88, (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[25] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *2016 IEEE International Conference*

on Software Maintenance and Evolution (ICSME), October 2016. 411–422. https://doi.org/10.1109/ICSME.2016.35

[26] Sheng Liang and Gilad Bracha. 1998. Dynamic class loading in the Java virtual machine. *SIGPLAN Not.* 33, 10 (October 1998), 36–44. https://doi.org/10.1145/286942.286945

[27] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 2013. Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM '13)*, November 08, 2013. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/2516760.2516769

[28] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-Time Configuration Options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, September 15, 2014. Association for Computing Machinery, New York, NY, USA, 445–456. https://doi.org/10.1145/2642937.2643001

[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, May 31, 2014. ACM, Hyderabad India, 1013–1024. https://doi.org/10.1145/2568225.2568229

[30] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses. *Empirical Software Engineering* 27, 1 (January 2022), 16. https://doi.org/10.1007/s10664-021-10013-5

[31] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. 2021. A Large-Scale Empirical Study of Android App Decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2021. 400–410. https://doi.org/10.1109/SANER50967.2021.00044

[32] Jean-Marie Mineau and Jean-François Lalande. 2024. Evaluating the Reusability of Android Static Analysis Tools. In *ICSR 2024 - 21st International Conference on Software and Systems Reuse (LNCS)*, June 2024. Springer, Limassol, Cyprus, 153–170. https://doi.org/10.1007/978-3-031-66459-5_10

[33] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 77–88. https://doi.org/10.1109/ICSE.2015.30

[34] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component communication mapping in android: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013. 543–558.

[35] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, October 26, 2018. ACM, Lake Buena Vista FL USA, 331–341. https://doi.org/10.1145/3236024.3236029

[36] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2018. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. (2018).

[37] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. The Internet Society.

[38] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys* 49, 3 (October 2016), 1–30. https://doi.org/10.1145/2996358

[39] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, February 15, 2014. ACM, Orlando FL USA, 143–153. https://doi.org/10.1145/2544137.2544159

[40] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. 2014. Information Flows as a Permission Mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, September 15, 2014. ACM, Vasteras Sweden, 515–526. https://doi.org/10.1145/2642937.2643018

[41] Dennis Titze and Julian Schutte. 2015. Apparecium: Revealing Data Flows in Android Applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, March 2015. IEEE, Gwangiu, South Korea, 579–586. https://doi.org/10.1109/AINA.2015.239

[42] Akihiko Tozawa and Masami Hagiya. 2002. Formalization and Analysis of Class Loading in Java. *Higher-Order and Symbolic Computation* 15, 1 (March 2002), 7–55. https://doi.org/10.1023/A:1019912130555

[43] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, November 07, 2014. ACM, Scottsdale Arizona USA, 39–50. https://doi.org/10.1145/2666620.2666630

[44] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *ACM SIGSAC Conference on Computer and Communications Security*, November 2014. ACM, Scottsdale Arizona USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[45] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and René Rydhof Hansen. 2014. Formalisation and Analysis of Dalvik Bytecode. *Science of Computer Programming* 92, (October 2014), 25–55. https://doi.org/10.1016/j.scico.2013.11.037

[46] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective Real-Time Android Application Auditing. In *2015 IEEE Symposium on Security and Privacy*, May 2015. IEEE, San Jose, CA, 899–914. https://doi.org/10.1109/SP.2015.60

[47] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 89–99. https://doi.org/10.1109/ICSE.2015.31

[48] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, March 02, 2015. ACM, San Antonio Texas USA, 37–48. https://doi.org/10.1145/2699026.2699105

[49] Wenwen Zhou, Yang Yongzhi, and Jiejuan Wang. 2022. Dynamic Class Generating and Loading Technology in Android Web Application. In *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, July 2022. 1–6. https://doi.org/10.1109/ISNCC55209.2022.9851782

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

CentraleSupélec

Titre : TODO 24 ▶ *Find a title* ◀

Mots clés : Android, analyse de maliciels, analyse statique, chargement de classe, brouillage de code

Résumé : Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque.

Title : TODO 25 ▶ *Find a title* ◀

Keywords: Android, malware analysis, static analysis, class loading, code obfuscation, TODO 26 ▶ *More Keywords* ◀

Abstract: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque.