UNIVERSITY OF CAMBRIDGE

Security metrics for computer systems

Daniel R. Thomas

Peterhouse

A thesis submitted for the Degree of Doctor of Philosophy

Abstract

Over the last four years the majority of Android devices have been susceptible to known critical vulnerabilities for extended periods of time. Technical solutions to improve security have been proposed, deployed and have helped. However, they have not stopped Android having known vulnerabilities. The root cause is the lack of prompt delivery of security updates to devices, which, in turn, is due to a lack of economic incentives for manufacturers to prepare, test and deploy security updates. A critical prerequisite for such economic incentives is robust metrics for the relative security provided by different manufacturers.

To measure the security of Android, data on vulnerabilities in Android is required. Chapter 3 describes the creation of a database of critical Android vulnerabilities and Chapter 4 combines this with data on Android devices to determine the proportion of Android devices that are vulnerable. On average, 88% of devices were vulnerable to one or more critical vulnerabilities between July 2011 and March 2016.

By modelling the deployment of updates on Android, Chapter 5 shows that fixing some design flaws takes 5 years. While technical solutions, including those provided by Google Play Store, have protected devices from some vulnerabilities, devices remain exposed to design flaws allowing remote code execution. Hence, technical solutions are insufficient and prompt security updates are required.

Chapter 6 proposes new metrics to measure the security of computer systems, including the composite FUM score. Using this score to measure the Android ecosystem reveals that while all devices perform poorly, some manufacturers are significantly better than others. The Android average is 2.71 out of 10, Nexus devices score 5.63, and LG is the best manufacturer with a score of 4.28. Chapter 6 also suggests that manufacturers, rather than users or operators, are responsible for the lack of security updates.

Computer security today requires both technical solutions and economic incentives. Android security is currently poor, despite numerous technical advances. This dissertation explores how economic incentives could help and proposes the FUM metric to provide customers, regulators and corporate purchasers with the data they need to make more informed decisions.

Declaration

This thesis:

- is my own work and contains nothing which is the outcome of work done in collaboration with others, except where specified in the text;
- is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other university; and
- does not exceed the prescribed limit of 60,000 words.

Daniel R. Thomas
September, 2015

Acknowledgements

This work was supported by a Google focussed research award; and the EPSRC [grant number EP/P505445/1].

Thanks to Richard Clayton for insight and useful feedback. Thanks to Laurent Simon, Thomas Coudray, Adrian Taylor, Justin Case, Giant Pune and Khilan Gudka for reporting vulnerabilities. Thanks to the anonymous reviewers at SPSM for their insightful comments.

Thanks to my examiners Andrew Martin and Robert N. M. Watson for extensive and useful feedback.

Thank you to Jeunese Payne and Diana Vasile for help with proof reading.

This work would not have been possible without the substantial engineering and research efforts of Daniel T. Wagner in the Device Analyzer project and of the many thousands of contributors of data to the Device Analyzer project.

The Rwanda data was supplied by Sherif Akoush and Ripduman Sohan.

Discussion with Andrew Rice, particularly about the ideas presented in Chapter 6 has been very helpful.

Throughout my PhD the support of Oliver R. A. Chick and the SN14 team has been most helpful.

Alastair R. Beresford has been a brilliant supervisor, and a great help.

My thanks to my wife Kirsty who married a PhD student, supported and encouraged me to finish on time.

Soli Deo gloria
Glory to God alone

Contents

CHAPTER 1

INTRODUCTION

In 1950, there were 5 computers in the world; by 2010 this had risen to 1.5 billion, and in affluent mature markets there are now several computers for each person including laptops, tablets, and phones (§A.1). With the progression towards the Internet of Things (IoT), Gartner expects there to be 500 smart devices in affluent mature market homes by 2022 [114].

The security methods used to protect computers have also changed. In 1950, there were no computer networks, but in 2010 there were 13 billion Internet connected devices [98]. This network connectivity exposes devices to external attack. Devices often have web-based interfaces using JavaScript and are subject to several classes of vulnerability, such as SQL injection and cross-site scripting; and must be protected with defences, such as firewalls and input escaping libraries.

New approaches have also been required due to the change in the way software is distributed and produced. Originally software was obtained by borrowing the paper tape from the tape library [257], but, over time, more convenient methods such as magnetic tape, CDs, and downloaded executables have become successively pervasive. Commercial operating systems such as Android and iOS now support app stores, which allow for the convenient distribution of software (though not with the flexibility that Debian achieved in 1996) with 4 million apps available from mobile app stores in 2015 [16]. The maliciousness of software is determined by the developers who make it. In 2014 there were 19 million developers [259], many of which are small operations, writing software for app stores, with little reputation to lose and operating out of jurisdictions where prosecution may be difficult. Since stores allow the convenient distribution of untrustworthy code, stores often ensure the code runs in some sort of sandbox, which restricts what the code can do. Unfortunately, vulnerabilities in

these sandboxes that allow privilege escalation can bypass this protection and so running untrusted code remains dangerous.

These new problems and solutions will become more important as devices become more powerful and numerous and the IoT takes off. The smart devices that compose the IoT could improve security by enabling systems like Pico [217], which would provide secure authentication, unlike earlier systems such as MIFARE, which are insecure [112]. However, early indications are that it will be fraught with security problems. Attacks have been found against Smart TVs [181]. Cars can be remotely compromised [146] and the instructions on how to do so have been published [170]. Reports of remote attacks on cars over the Internet [128], have triggered the recall of 1.4 million cars. Large-scale analysis of device firmware has found many vulnerabilities [58] and even fridges have been compromised and used to send spam [203]. The lack of security updates is particularly problematic. For example, a Linux worm was discovered which attacked IoT devices that had not been updated for a vulnerability in PHP that was fixed 18 months earlier [120].

If the mean time before replacement of devices is less than the mean time before vulnerabilities are discovered then, on average, security updates will be required. If there are 500 devices in a house, then the probability of security updates being required will be high. Microsoft has had a program to improve the security of its software for a long time and yet still issues around 100 security advisories a year, and issues security updates every month [143]. Therefore, products that are used for more than a few months will require security updates. This is expected, as many IoT devices, such as fridges, are white goods and are expected to last 7–10 years [45], or building components, such as light switches, that are expected to last 30–60 years [164]. This long device lifetime runs contrary to the trend in the replacement times for computers. University mainframes were replaced after 7.3 years, PCs 5.8 years, and smartphones only 2.0 years (§A.2). However, as the IoT leads to hundreds of devices per person, devices must last tens of years or users would spend too much time replacing them (it would also be financially and ecologically irresponsible). This goes against the trend of shorter device lifespans rather than longer ones.

Even if updates are made available, scaling the approval of the installation of updates to hundreds of devices per person is difficult, and it will not work if this requires manual approval. In Windows 10, updates will be automati-

cally installed without approval [12]. Even attempting to authenticate control of these devices will require scalable strong authentication [218] and other new techniques to manage the many devices.

Many IoT devices are controlled via services running in the cloud. These cloud services are easier to manage than end-user devices because the cloud services run on virtual machines [256] or on app frameworks. This gives the provider more control and a simplified interface and so the services can quickly be replaced [163], are easier to update, and can even be updated at runtime [95]. Both on physical devices and in the cloud there has been a change in who is in control of data and behaviour. Increasingly it is the developer or provider rather than the user that controls the data collected and the behaviour of the device. Users have to pick their feudal lord and swear allegiance to them in return for the services the user receives [209].

Google is one such feudal lord; the operating system used for many of the IoT devices in their domain is Android. In particular since smartphones and tablets are key to interacting with new IoT devices (as opposed to existing IoT devices like desktops, tablets and smartphones) and with over 80% of mobile device sales in 2014 Android is already the main platform for interfacing with the IoT. Android is the largest mobile operating system,[1] running on 1.6 billion devices in 2014 [214] with 1.6 million apps on the Google Play Store [16] developed by 513 thousand developers in 2015 [15]. Due to its open nature, it is used on many new devices. This openness and flexibility comes at a cost: there is a long software update pipeline, from when vulnerabilities are fixed in updated upstream open-source software, which is incorporated by Google, then by device manufacturers, then tested by network operators before being installed by users (§6.1). Other operating systems have shorter pipelines: Microsoft fixes vulnerabilities in Windows and ships them to users. Apple incorporates upstream fixes into iOS but can ship them to users, and Ubuntu and Debian incorporate upstream fixes but then ship directly to users. Hence, understanding how this long pipeline affects the security of Android and developing techniques to improve it will illustrate problems and solutions, which may apply to the IoT.

Information on the security of the IoT at scale is not available today as it does not yet exist. While there will be new problems in building a distributed

[1] It might also have been the largest computer operating system.

system of things [161], information on Android does exist and lessons learnt from studying Android are likely to be applicable to the IoT. Some existing IoT devices – such as fridges, ovens and toasters – use Android and with its widespread deployment on mobile devices, Android devices may constitute the largest segment of current IoT devices. The IoT will be built by many innovating companies as many products will fail and there will be many niches and so the flexible and distributed nature of the Android ecosystem might be a good fit. Open-source software is cheaper to produce, and many existing IoT devices use a Linux derivative like Android. While many IoT devices may be embedded devices that don't run operating systems like Android, those devices will still be controlled by devices that do have the capability to interface with humans that operating systems like Android provide. Hence understanding how to improve the security of the Android ecosystem should help prevent the same problems occurring at a larger scale in the IoT.

How can improved security for the IoT be encouraged? Lessig suggests that there are four methods for affecting behavioural change: the market, architecture, the law, and social norms [154]. Market and architectural forces act before something takes place, preventing it from occurring, by making it prohibitively expensive or physically impossible. They are also rigid and cannot be bypassed, even with justification. Legal and social norms result in consequences and the knowledge of those consequences causes people to choose not to do forbidden things. However, legal and social norms are flexible, for good or ill: with a sufficiently worthwhile reason (from the perspective of the individual) they can be ignored. In an emergency, a driver can break the speed limit to get a casualty to hospital but if their car has a speed limiter or the road has speed bumps then they cannot. The legal and social norms are more flexible. In an emergency the driver can speed, but the architectural solution is rigid and cannot be changed in an emergency, but also prevents speeding more comprehensively.

These ideas can be applied to security: For market forces, if consumers cannot determine the security of a product, the asymmetry of information means that the consumers are in a lemons market [5] and so cannot pay for security as the consumers do not know which products provide it. Therefore, there is no incentive for manufacturers to provide it. For architectural constraints, if there is pressure to improve security, then architectural solutions to provide it will be found, such as making WebView updatable through the Google Play Store in

Android 5.0 [124]. For legal norms, before regulations can be made about security on devices, and compliance can be evaluated, security must be measured. For social norms, if manufacturers cannot determine how they compare with each other, and their peers cannot tell if they are doing badly, there will be no social pressure to improve. Hence, there is a requirement for metrics to measure the relative security provided by different device models and device manufacturers. This allows regulators, consumers and other manufacturers have the data they need to cause changes in behaviour, which will improve security. I propose a metric to provide some of these data and use it to evaluate Android device models, device manufacturers, and network operators with respect to updates for security vulnerabilities in Chapter 6.

While there has been less work on the impact of market forces, and legal and social norms on security; many architectural solutions to improve security have been proposed:

- Reducing the trusted computing base, through methods such as library operating systems where all functionality that is not explicitly used is compiled away, such as with Unikernels [162].[2]

- Formal verification of software to verify the absence of vulnerabilities.

- Reducing the external attack surface through minimal APIs (e.g. in hypervisors), firewalls, and compartmentalisation [251].

- Using VPNs and TLS to bypass risky parts of the network and so avoid attack (though this does not prevent inference attacks based on traffic patterns [225]).

These solutions are brittle: either they prevent the attack or they are bypassed.[3] They do not have the flexibility that other forces can provide. The market, legal and social forces would be empowered by the provision of comparable data on the security of devices, and so that is the focus of this dissertation.

[2] A Linux VM might only be intended to use TCP but might still be exposed to vulnerabilities in the kernel's implementation of SCTP. With a full analysis of the whole program and environment a Unikernel can remove all functionality that is not required for correct operation, excluding kernel or library features that exist to support other applications or platforms.

[3] If there are multiple layers of these solutions then each layer may need to be bypassed separately.

1.1 Dissertation outline

The next chapter presents background research. This dissertation investigates both implementation flaws such as buffer overflows (Chapter 3) and design flaws that require a two-sided fix (Chapter 5). It shows that the deployment of fixes for both is slow due to lack of updates and proposes a model for predicting the deployment of updates. To evaluate the security of Android that results from this slow update process, a database of vulnerabilities is required, and the database and the collection process used is explained in Chapter 3. By combining this database with Device Analyzer data I evaluated the vulnerability of the Android ecosystem. Malicious code can be stopped by the store refusing to distribute it, by install time checks, or by runtime protections. Chapter 4 shows that it is the difficulty in getting malicious code onto devices – the protection provided by the store – that protects Android. Most Android devices are exposed to known critical vulnerabilities, which are exploitable if malicious code is able to run on the device.

However, there are vulnerabilities that would allow malicious code to run on Android devices via network attacks, which are fixed slowly (Chapter 5) and so relying on the difficulty of malicious code reaching devices is not sufficient. Hence, I propose the FUM scoring metric, which uses information on known vulnerabilities and the distribution of deployed versions, to evaluate the security of provided by different device manufacturers and use it to compare them, showing that there are clear differences between manufacturers (Chapter 6), motivating them to improve.

1.2 Publications

During the course of my PhD these papers have been accepted for publication:

- "Better authentication: Password revolution by evolution" with Alastair R. Beresford at Security Protocols 2014 [233].

- "The lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface" with Alastair R. Beresford, Thomas Coudray, Tom Sutcliff and Adrian Taylor at Security Protocols 2015 [239].

- "Security metrics for the Android ecosystem" with Alastair R. Beresford and Andrew Rice at SPSM 2015 [236].
- "Incentivising software updates" with Alastair R. Beresford at the Internet of Things software updates workshop 2016 [234].

The work in the latter three papers is presented in various parts of this dissertation. My contributions and those of others are distinguished in the respective chapters.

Dataset

Much of the raw and processed data and source code used in this dissertation is available [237, 238], excluding that which might identify individuals. Data from Device Analyzer [247] and AVO [232] used in this dissertation is already available. Licensing restrictions prevent the release of the Rwanda and FTSE data used in §4.3.

CHAPTER 2

BACKGROUND

> What has been will be again,
> what has been done will be done again;
> there is nothing new under the sun.
>
> Ecclesiastes 1:9 (NIV)

This chapter describes the relevant existing work, market conditions, and the source of data used later. It explores the threats users face, some of the motivating applications for secure mobile systems and the difficulties in correctly building secure systems. It describes the mobile market from 2001–2016, the behaviour of the Android OS and the Device Analyzer project for understanding the behaviour of Android devices. Then, since protecting users requires understanding users, it describes work analysing what users understand, how users behave, and what techniques work to improve understanding and behaviour. User behaviour is determined in part by the advice users are given, for example, users are told to install updates regularly. But there are a variety of update mechanisms – some with their own security problems.

Users need to install security updates because of vulnerabilities in the systems they control, there are metrics for measuring vulnerability, and studies of vulnerabilities affecting different systems. The economics of vulnerabilities play a key part in determining how vulnerabilities are fixed, what effect vulnerabilities have and even on whether security experts should try and find them, either manually or automatically. These vulnerabilities matter because of the malware that tries to exploit them to attack users' devices, so if security software could detect all malware and block it then vulnerabilities would not be a problem. There have been many efforts to detect malware but doing so is hard, particularly doing so without high false positive rates. Another strategy to prevent malware doing

damage by exploiting vulnerabilities is greater containment, which has led to work using various virtualising, sandboxing and policy enforcement techniques to try and stop malware using vulnerabilities, some of which have even been widely deployed.

Hence, there are four layers of defence, user behaviour in avoiding risky situations, prompt deployment of updates to fix vulnerabilities, accurate detection and blocking of malware, and robust containment and sandboxing of code. None of these layers is perfect and it is the security of the combination which determines the security of the system.

The reason why Android is subject to vulnerabilities is that it has a large API and hence a large attack surface. Approaches such as Unikernels that rely on the isolation properties of a hypervisor with a much smaller and simpler API than a kernel have better isolation guarantees [162]. While Android security is much better than the traditional desktop OS security that it built on, it still has not achieved the Kilimanjaro effect, it has not provided such a strong and sudden increase in security that malicious actors have abandoned trying to do bad things with it [53], however perhaps iOS has succeeded through imposing more restrictions faster.

Despite all this effort to characterise and improve the security of user devices, Chapter 4 shows that Android users are still exposed to substantial risk, mostly for economic reasons and due to lack of comparative data, which is provided in Chapter 6.

2.1 Threats – Who is attacking what?

To evaluate whether a particular security measure is useful or possible one must understand the threat that it defends against and the attack surface. The attack surface is the range of different places against which an attack can be mounted; what these are depends on the context. All the external facing parts of programs that you depend on, or running on machines you care about, form part of your attack surface from a cyber-security point of view. In the same way, the doors, windows and walls of your house form part of an attack surface from a physical security point of view. With your house you can probably enumerate the complete attack surface and have some understanding of where the weaknesses

are. However with cyber-security this is much more difficult and you probably have less visibility into what it is and how good it is, as many parts of it are not under your direct control. As Leslie Lamport said "You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done" [10]. Similarly when the compromise of a computer you have never heard of compromises your security then you know you have a distributed attack surface. As more of our attack surface moves to the cloud, the extent to which it is distributed increases and we find it less transparent and less under our control. This might be a good thing as perhaps large companies are better at security than we are [209]? On the other hand, large companies are more likely to be involved in schemes like PRISM and so the NSA and collaborating agencies might be hoovering up all your data [129].

Understanding the behaviours and intentions of likely attackers makes the risks posed by different attacks more concrete. There are many different kinds of attackers with increasing levels of resources: bot herders, phishers, spear phishers and APTs. The botnet herder does not care who you are, herders just want control of your machine, herders also do not have much resource to invest in an individual attack and so will send the same attack to millions of machines and not care about those who are not compromised. Similarly, phishers mostly send the same attack against many users, but phishers may also target it at particular groups such as companies or universities and tailor it to that group to make it more convincing. Spear phishers take this further, constructing attacks targeted at specific users such as CEOs and are willing to put significant effort into this. There is another class of attackers above this – the Advanced Persistent Threat (APT), which is where an organisation such as a state actor is willing to put huge time and effort into compromising your systems. Numerous news reports claim that many countries appear to have divisions responsible for doing this including the USA [178], China [117] and Israel [8]. Larger cyber-criminal organisations may be able to undertake similar kinds of attack. Even dissident groups or teenagers may attempt this kind of attack. While countries with secret nuclear programs may need to worry particularly about APT [151], ordinary people may also be caught in the crossfire [100]. These threats have become more concrete and disturbing as actual examples have come to light. PRISM reveals that the NSA and collaborating agencies act as a global passive adversary with access to data stored on cloud servers [129]. On an even more disturbing

level, the Syrian government has industrialised the process of torturing citizens for passwords for Facebook/Twitter etc. out of rebels and then attempting to social engineer all their contacts [32]. Designing systems that work even in the face of such powerful adversaries remains an open research problem.

How can these threats be mitigated? Naïvely you might think that examining all the source code and schematics for your system might verify that it contains no trojans and that it is secure. Unfortunately, even if an organisation had the resource to attempt it, there are many ways that trojan compilers or machine code could be used to insert flaws into compiled binaries that then persist in future compilations [240]. Additionally, finding all the security flaws in a large system is essentially impossible because there are always more bugs and fixing one is likely to introduce another (§2.6.2).

Instead, many systems try to reduce the attack surface by using the principle of least privilege [205]. For example, there have been efforts to reduce the attack surface of the Xen hypervisor system by breaking up the trusted components into smaller pieces with less responsibility (§2.9.1) [56]. The Certificate Authority (CA) infrastructure for Transport Layer Security (TLS) has a particularly large attack surface and exposes every user of secure websites to flaws in up to ~600 organisations [52] (§2.2.2). One solution to this is to use DANE to root trust at the top of the DNS hierarchy and use DNSSEC for secure delegation [182] This could significantly reduce the number of trusted organisations and hence the attack surface.

2.2 Secure apps require API security

There are security critical tasks for which computers, particularly mobile computers are vital. For example, authentication systems to replace passwords [34] such as my OTTA proposal [233] or Octokey [145] rely on a secure computing platform that users carry with them to store keys and execute protocols. However even with a secure platform, the system itself (protocols and their implementations) must be secure and writing correct protocols and APIs is hard.

Application Programming Interface security has been an active area of research since the 1980s. It is concerned with APIs used by developers and provided by libraries and hardware. Designing secure APIs is both difficult and of

fundamental importance to the security of the apps that use them. Chapter 5 presents an API failure and a model of the deployment of fixed APIs. This section discusses the different ways and levels at which APIs have been found to fail and the solutions proposed.

2.2.1 Failures

Designing APIs for security is difficult and many previous attempts have failed, some of those failures have even been deliberate [210]. Disasters have occurred at different levels of cryptography APIs. Even seemingly simple cryptographic primitives can be sensitive or brittle. For example, in 2008 a Debian developer patched OpenSSL's pseudo-random number generator to fix Valgrind warnings and weakened it such that it only generated 32,767 possible keys [4]. Similarly, exposing developers to raw hash functions such as MD5 and SHA may result in incorrect usage. In 2009, Facebook ended up using a four-digit random number to protect private photo albums [33]. As a result, Facebook developers now receive compile warnings if they use the standard PHP functions for MD5 or SHA rather than the higher-level HMAC functions provided by a separate Facebook library.

Problems abound in secure key management. In January 2013, a large number of private SSH keys were found on public Github repositories, as many users had decided to publicly share the configuration data stored in the home directories of their Linux and Unix machines but failed to redact the contents of their `.ssh` directory. In June 2013, I found that Citrix had accidentally published private SSH keys that allowed access to their network on Github when Citrix open sourced XenCenter. Citrix resolved this quickly.

Libraries for secure communications are also hard to use. A survey of Android apps [99] found 8% (1 074) of the 13 500 apps scanned contained TLS code vulnerable to man-in-the-middle attacks, 14% of the 7 690 that were using TLS. This is because the Android library is hard to use correctly, particularly when the developer uses a self-signed certificate during development. Session management on the web is also an area of concern. Many languages and runtimes lack a standard method of session management, resulting in developers writing their own implementations using cookies. These ad-hoc implementations typically have weaknesses such as leaking the key used to generate session

cookies and failing to require periodic re-authentication, despite well-known solutions to these and other problems [173].

APIs that are large and complex are likely to suffer from vulnerabilities. This arises because API calls are typically chained together, resulting in a large number of composite operations that may or may not be secure. A good example in this space is EMV ("Chip and PIN") security for authenticating debit and credit cards to point-of-sale terminals and automated teller machines. The API for EMV is huge, and many vulnerabilities have been found. Attacks involve exploiting a poor choice of encryption methods to leak a key and confusing a hardware security module by combining different API calls to leak secret keys [9]. Similarly, the introduction of card readers for online banking has been beset by problems [85]. Online payment has also been beset by the '3D-Secure' debacle, which is impossible to distinguish from a phishing attack and fails in many ways [174]. Banks have moved away from using it to perform any authentication checks. OAuth, a standard for authorisation, commonly used for authentication is so complicated to use that it has required a study to systematically understand common incorrect uses of it and to explain correct usage [48].

In summary, existing libraries are frequently difficult to use. Existing libraries require lots of setup and configuration, while sensible defaults or high-level APIs, which use low-level components in suitable ways, would be more appropriate. For example, it's not uncommon for an API to require seven method invocations to accomplish a task when a single high-level method would suffice [26, p6]. Errors also occur through the failure to check return values for error codes. This is such a common occurrence in OpenSSL that there have been attempts to automatically detect failure to check return values [152]. Libraries also implicitly assume that the developer will undertake proper key management and do not provide library support to prevent dangerous use of keys [9].

2.2.2 Levels in the cryptography stack

To understand how all these different failures fit together it is helpful to classify the range of different parts of APIs into different levels in a stack. Cryptographic operations can be classified into six levels, each of which depends on the previous

one:[1]

1. Mathematical foundations: This includes big integer operations such as `modpow`, arithmetic, secure random number generation and vector operations.

2. Cryptographic primitives: This includes block ciphers (AES) and modes of operation (CTR, GCM), public-key cryptography (RSA), hash functions (SHA), message authentication codes (HMAC) and key derivation (PBKDF2).

3. Cryptographic functions: This involves correctly using the primitives with the correct parameters to produce functions such as `sign`, `encrypt` and `encryptDeterministically`.

4. Key management: This involves public-key infrastructure, key rings, certificate authorities, webs of trust and bootstrapping.

5. Secure communications and session management: Allowing two entities to communicate securely over an untrustworthy channel (e.g. SSH, SSL) and keeping track of which users are logged in and who is authenticated to whom in a secure manner (e.g. cookies).

6. Secure Remote Procedure Calls (SRPC): Operations like `login` and `register`, protocols like EMV.

Existing flaws occur because either: (1) a library implementation is incorrect; (2) developers use an API at an incorrect level for the job at hand; (3) the library API is too hard to use or fails to select suitable defaults for the developer; and relatedly, (4) the library APIs are too complex for the designer or researcher to reason about or consider as a whole.

The Debian key fiasco described above is an example of failure at level 1 and came about due to an incorrect library implementation. Facebook's poor protection of private photo albums occurred because developers used level 2 primitives, when Facebook developers should have used level 3 primitives. EMV's overly complex implementation at level 3 resulted in the leak of keys. Incorrect use

[1]This classification is the result of a collaboration with Alastair R. Beresford and has not previously been published.

of TLS and insecure cookies are both examples of failures at level 5 and were caused by the lack of simple secure APIs to check certificates properly and the failure to use, or lack of support for, session management.

TLS is a level 5 protocol and has a rich history of failures at all layers in the stack beneath it [52]. In addition to the usability flaws and the efforts to automatically find flaws in OpenSSL, both discussed in §2.2.1, there are many other issues. Level 1: The Netscape browser used a weak random number generator to make keys [118]. 0.5% of TLS certificates have been found to have recoverable private keys due to poor seeds for their random number generators on embedded devices [131]. Level 2: Timing attacks on RSA [38] and ECDSA [37] decryption in insufficiently careful implementations resulted in the leak of information about the key. Level 3: CBC mode does MAC-then-encrypt rather than encrypt-then-MAC and so decryption is attempted on unverified data, this has resulted in various timing attacks that leak the key [88]. Additionally, a predictable IV is used for all records except the first, which is insecure [21]. Level 4: The CA infrastructure for TLS is widely considered to be broken due to the sheer number of entities trusted completely (§2.1). Several of them have been compromised resulting in incorrect certificates being issued and others are run by untrustworthy state actors or simply by companies who should not have that power. Marks and Spencer (a UK supermarket chain) have that power [89] and so could issue wildcard certificates for "*.*". In particular, if your threat model includes APT, then you cannot rely on the CA infrastructure. Key rollover is also difficult [140] and frequently a manual process. Even large companies sometimes allow important certificates to expire without finding replacements [153].

2.2.3 Proposed solutions

Better libraries

There have been various attempts to improve the state of the usability of cryptographic libraries and to try and prevent developers making mistakes. In my previous work on Nigori [235] (a system for storing secrets in the cloud) I used the Stanford JavaScript Crypto Library (SJCL) [219], which implements functionality at levels 1, 2 and 3.

However, I experienced some difficulty in using it because, in an effort to pre-

vent misuse of level 2 functionality, it was hard to use for my purposes. Nigori had uncommon constraints (e.g. it needed deterministically generated IVs, which are safe to use in this case but not in general) which meant that the standard functions from level 3 were inappropriate. This is an example of architectural mismatch [113]; two different levels of API need to be provided by a library, the simple public functions that just do the right thing and the expert API that customises the functions implementation.

Keyczar [82] recognises that merely implementing the first three levels is insufficient because much can go wrong with key management (level 4), in particular key rollover is hard. As a result, Keyczar provides functionality at levels 3 and 4. The Networking and Cryptographic library (NaCl) [26] seeks to provide functionality at 3 and 5, particularly emphasising speed and safe defaults, but it lacks functionality at level 4.

Formal methods

Despite the many problems in TLS discussed previously (§2.2.2) it has a proof of correctness [186]. Proving protocols to be secure is a useful way of finding problems in them and protocols that lack them are treated with caution. Unfortunately just because a protocol has been proven to be secure this does not mean that it is secure. Assumptions may be violated, the property proved may not be the one actually required, and implementation flaws and side channels abound. The first attempt at formal proof of protocols was the BAN logic [43], which formalised the need to prove the freshness of messages and allowed reasoning about the belief that different parties could have given certain starting assumptions and the protocol. This was extended in the GNY logic [119] to cover a wider range of protocols and to better capture the expectations of the protocol designer. This and other work was later unified to also support the concept of time [227] at the expense of a more unwieldy system.

2.2.4 Fixing API vulnerabilities

Once an API vulnerability has been found and a fix has been created, this fix must be deployed. Since this fix may have changed the API in a non-backwards compatible way the process of deploying the fixed API may be slow as the new API needs to be used by both the clients and the servers before it is fully deployed.

A model for the deployment of new API versions is proposed in Chapter 5 along with a case study of an API vulnerability.

2.3 Mobile operating systems

Secure protocols need to run on secure platforms, such as secure mobile devices. This section provides context on the mobile market during the period relevant to this dissertation. In 2013 the key mobile operating systems were Android, iOS, Windows Phone, BlackBerry and Symbian. The percentage of total smartphones sold by operating system between 2008 and 2014 is given in Figure 2.1 and the absolute sales figures in Figure 2.2. These show the dramatic growth in smartphone sales and how Android has come to dominate the market with over 80% of sales in 2014. The data for these plots for 2008 and 2009 come from Gartner, for 2010-2013 come from Statsia [222], and for 2014 from Strategy Analytics [166].

Nokia's Symbian S60 was first released in 2002 [226] and was abandoned in 2011 [90]. RIM's first BlackBerry smartphone was released in 2003 [1] and the operating system was discontinued in 2016 [50]. Microsoft has repeatedly restarted its mobile OS with Windows Mobile released in 2003 [81], replaced by Windows Phone in 2010 [202] which in turn was replaced by Windows 10 Mobile in 2015 [35]. Apple's iOS was first released in 2007 [216]. Google's Android was first released in 2008 [171].

In addition to the large proportion of devices that run Android natively, in 2011 BlackBerry gained support for running Android apps [133] (in September 2015 BlackBerry released a native Android phone [30]). This is because the usefulness of a mobile platform depends on the number and quality of apps available for it.

The vast majority of published research into mobile systems focusses on Android. This is partly because it has the vast majority of the market but additionally its open-source nature, greater flexibility and ease of modification (in comparison with iOS and Windows Phone/Mobile). While it would be possible to repeat work presented in this dissertation for other mobile operating systems, if the relevant data was available, since Android is so dominant, studying Android covers most mobile systems.

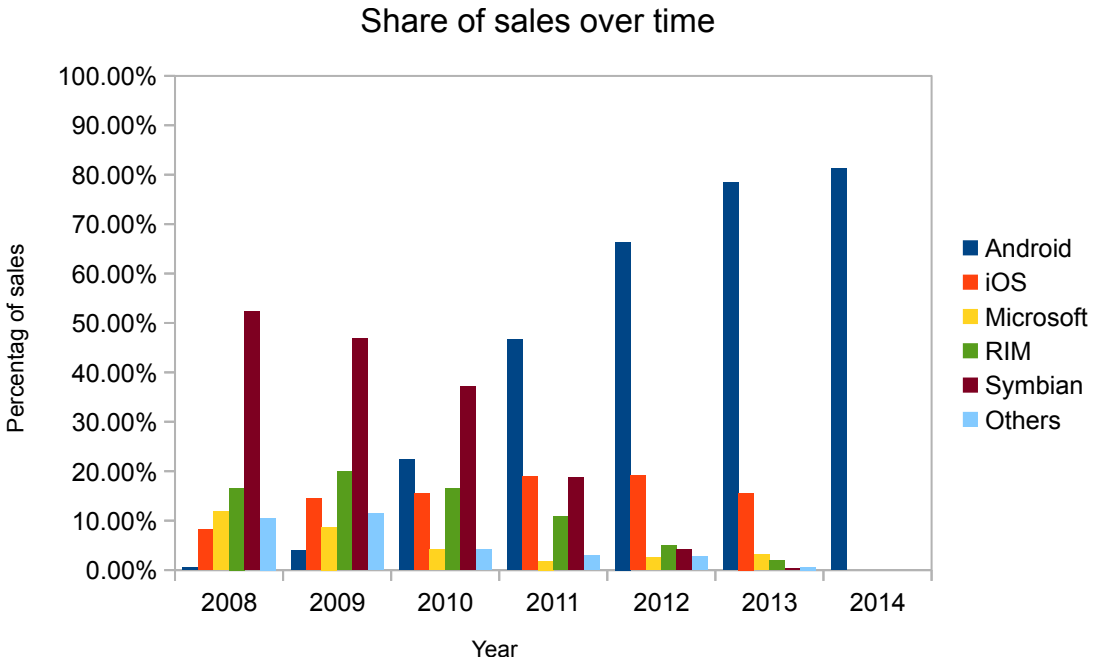


Figure 2.1: Percentage of smartphone sales by operating system

2.3.1 Key players in the mobile market

This section provides context on the players in the mobile market that will be referred to later. There are several groups of actors in the mobile market that have a substantial impact on it. Companies that produce the software, the hardware (device manufacturers), the chips, supply mobile network services and often sell the handsets (network operators), regulators, advocacy groups, and advisers.

Google, Apple and Microsoft produce respectively the Android, iOS and Windows Phone/Mobile operating systems though other companies also contribute, particularly to Android. Samsung, Nokia, HTC, Apple, LG, Motorola, Sony and Huawei all produce handset hardware (device manufacturers). Qualcomm produces chips used in handsets, particularly for radio/baseband purposes and contributes to Android. ARM designs the ARM processors used in most mobile devices, though Apple produces its own versions of the ARM processors. Microsoft bought Nokia who use Windows Phone/Mobile. O2, T-Mobile, Orange and 3 are network operators with a UK presence, Sprint, AT&T and Verizon are network operators with a US presence.

The European Union (EU) is an important regulator setting some tariffs for network operators and requiring mobile devices to support micro-USB charg-

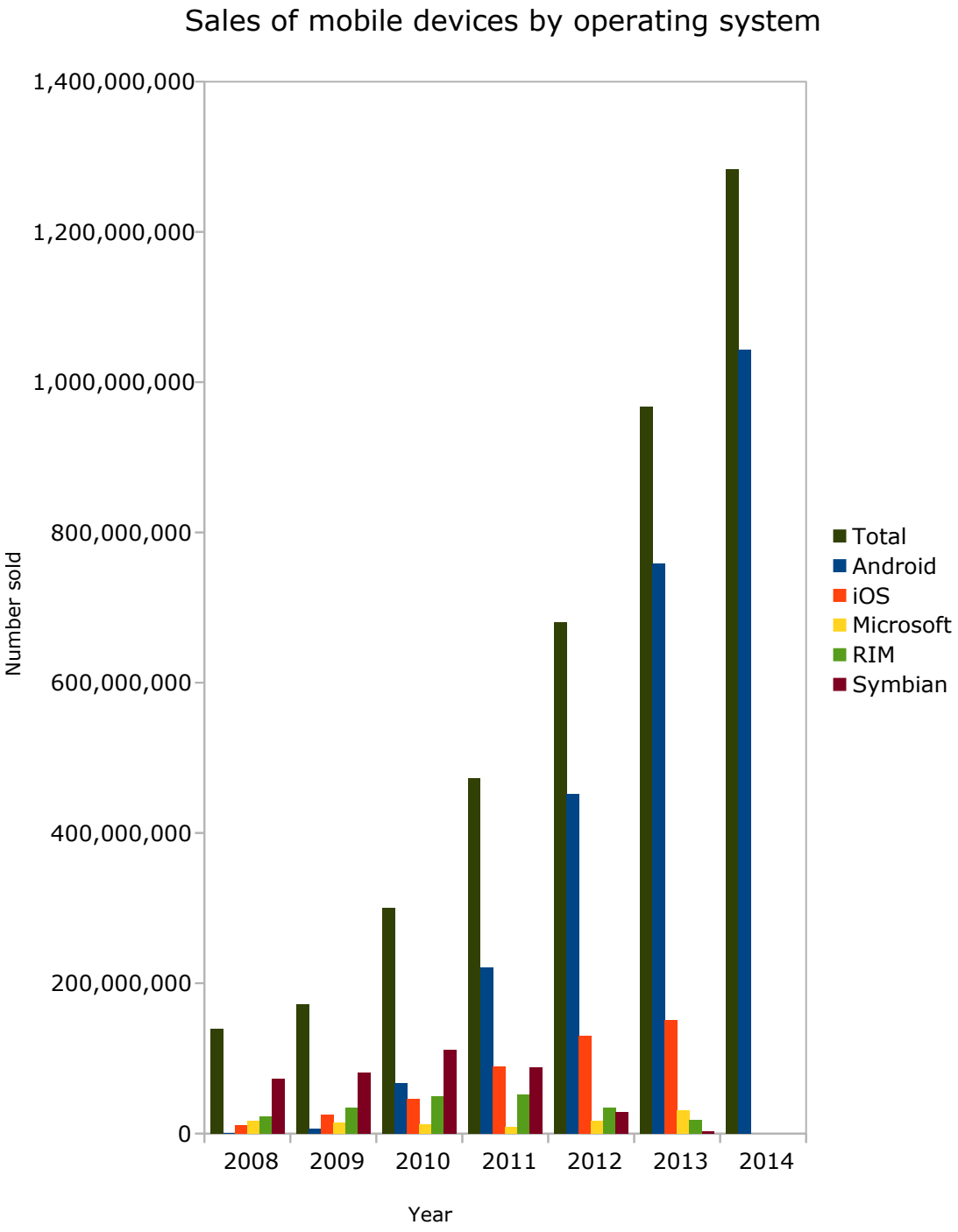


Figure 2.2: Number of of smartphones sold each year by operating system

ing. The USA's Federal Trade Commission (FTC) is another key regulator and has forced some manufacturers to supply updates at the request of the American Civil Liberties Union (ACLU). The UK's CESG provides advice to the UK government about about computer security and is part of GCHQ which is the UK equivalent of the USA's National Security Agency (NSA).

Google has produced its Nexus line of Android devices in collaboration with a number of device manufacturers. With HTC it produced the G1, Nexus One and Nexus 9; with LG it produced the Nexus 4, Nexus 5 and Nexus 5X; with Samsung it produced the Nexus S, Galaxy Nexus and Nexus 10; with Huawei it produced the Nexus 6P; with Motorola it produced the Nexus 6.

Further discussion of the ecosystem of companies involved in the production of Android is described in §6.1.

2.3.2 Mobile business models

There are several sources of revenue in the mobile market: sales of devices, contracts supplying network services, adverts (including monetisation of user data), revenue from app/media stores and revenue from in-app payments.

Apple sells devices and takes a 30% cut of the sale of apps from its app store and of in-app payments; it also sells access to content through iTunes. Google sells Android Nexus devices but most Android devices are sold by other companies. Google takes a 30% cut of the sale of apps and media content through the Google Play Store and from in-app payments. It also runs an advertising network which is used by many Android apps and Android uses other Google services such as Google search by default which then displays adverts. Despite Android devices having such a large share of sales, Apple's much lower sales figures for iOS still generates 92% of the profits for the entire smartphone industry. Samsung takes another 15% with the remaining device manufacturers making a loss [158].

2.3.3 Android protection model

Older operating systems attempted to protect users from each other and the kernel from the users. They did this by giving all code running on behalf of a particular user the same ambient authority. Android attempts a more fine-

grained per-app privilege separation. There is one Unix style 'user' per-app and apps have to request at install time the fine-grained permissions that they will use later. Before apps are installed, Google's Verify Apps software, essentially an anti-malware application, scans the app to determine if it appears to be malicious, and warns the user not to install it if it is found to be malicious. Verify Apps also scans existing apps once a week or once a day. If Google realises it has distributed malicious apps through the Google Play Store then it can retroactively remove them from devices. If apps listed on the Google Play Store are reported to be malicious then they are quickly taken down and through comments and ratings some reputation for the app and developer is exposed to the user. Before apps can are listed on the Google Play Store they are scanned by Google's Bouncer anti-malware program and the developer has to pay a small fee to register as a developer.

2.3.4 Android apps and API levels

Android apps are distributed as Android packages (APKs), which are structured zip files. These contain the compiled code and other resources such as image files. The Android framework is written in the Java language and the default language for Android apps is Java however they can also include compiled libraries of native code. One of the other resources is the manifest file that provides information about the app such as the permissions it requires and its entry points. It also specifies the target Android API version and the minimum API version. Android refuses to run apps that specify a minimum API version higher than it can provide and it maintains compatibility with existing apps by providing compatibility with the target API version of an app if the device is running a higher API version. An app developer ensures that an app only relies on features that were included in the API release corresponding to the specified minimum API version and tests that it works properly with the target API version.

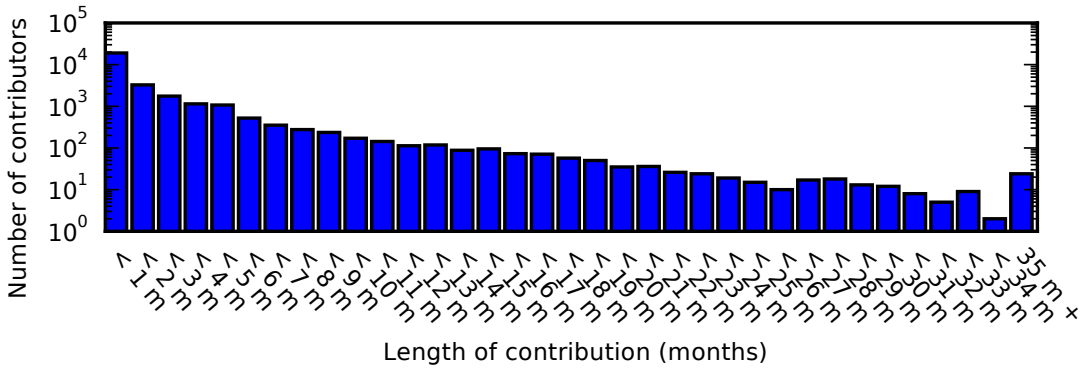


Figure 2.3: Number of contributions of different lengths (log scale)

2.4 Device Analyzer: Analysing the behaviour of Android devices

The Device Analyzer project [247] began in early 2010 and first produced useful amounts of data in July 2011. The purpose of the project is to collect data on the behaviour of Android devices in order to allow a wide range of research questions to be answered. It is not possible to run such an app on iOS as it deliberately lacks the relevant functionality. This is done through an Android app called Device Analyzer that runs on devices and collects data on their behaviour, carefully anonymising certain strings such as Wi-Fi SSIDs by hashing them with a key stored on the device, before periodically uploading the data to the Device Analyzer server in Cambridge. This data is then further processed before being released to researchers who have signed appropriate legal agreements to protect participants, three months after it was collected. The delay allows users to request that data be removed before it is published. Installation of the Device Analyzer app is voluntary and little has been done to promote it, though press coverage of the work presented in this dissertation, and collaborations with two network operators added thousands of contributors in particular time periods. Some contributors only contribute for a few hours or days while others contribute data for months or years. The histogram of the length of contributions is shown in Figure 2.3.

Device Analyzer collects data on many aspects of Android device usage [245] including: 'coarse grained' (network based) location data (the sharing of this is prohibited by default), the list of apps that are running, cell ids of mobile

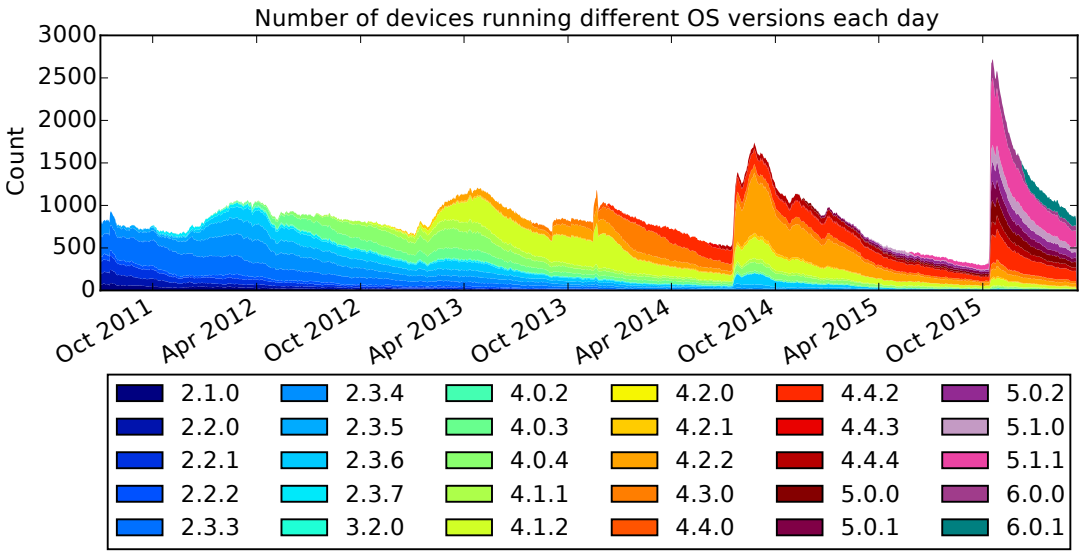


Figure 2.4: Number of devices in Device Analyzer running different Android versions each day.

networks, signal strengths, screen lock usage, battery levels and many others. In this work one of the sets of data that is used is the OS-version information. The number of devices in Device Analyzer running different versions of Android is shown in Figure 2.4. This stackplot also shows the total number of devices contributing to Device Analyzer each day. A normalised version of this plot is given in Figure 4.2 in §4.2.3. The spike in user numbers in August 2014 is explained in §4.3.3 and the spike in October 2015 is due to press coverage about this research resulting in increased Device Analyzer installs.

2.5 Users – Whom are we protecting and what are our expectations?

The security of computer systems usually depends on the humans who use or run them. These people are not the enemy but it is important to understand their limitations [3] and to build systems that correctly incentivise them [11]. Systems for security need to be built to be usable. Work on the usability of PGP for encryption and signing of emails found that the majority of users could not use it correctly within 90 minutes [254] and later work found that the situation had not improved seven years later [212]. Detecting phishing is another prob-

lem with usability as users cannot parse URLs to determine that the URLs do not point to the correct domains, and banks' marketing departments send out genuine emails with all the telltale signs of phishing in them [84].

2.5.1 Users do not understand permissions

On Android the permission system is used to allow the user to decide what an app is able to do and what data it has access to. However users do not understand Android permissions [144]. Only 17% of them pay attention to permissions and only 3% fully understand them [103]. Securacy aims to help with this by asking users what they are concerned about and then warning users about apps with permissions that concern them [104].

Location privacy has long been an area of concern. Approaches include mix zones where users change identities when users enter an untracked area [23], or using cryptographic protocols to perform privacy preserving location sharing [83]. Other mechanisms have been implemented for Android where the granularity and accuracy with which location data is reported to apps is carefully controlled [101]. In mobile computing, when location information at some resolution is always available, and protected by permissions, users need to understand what these permissions mean to make good choices. In particular, on Android users need to know the difference between precise and coarse-grained location, and were surprised to find out how precise 'coarse-grained' location is [110], and how often location information is collected [111]. If users are given the ability to revoke permissions or a adjust the granularity with which location information is provided to apps and then are told how apps are using the permissions then 58% of users further restrict permissions [6].

Since users do not understand permissions, there have been attempts to get computers to understand them so that users can be advised if the permissions requested are necessary. AutoCog attempts to measure the fidelity with which the requested permissions are explained by the app's description [196]. DroidJust tries to justify the permissions requested by analysing the behaviour of the app and what it uses the data for [49].

However, if an app requests a permission the user must either grant that permission or not install the app. MockDroid fixes this by allowing the user to change the permissions granted to apps at runtime and mocking access to the

app if the app has been denied permission [24]. So if the app tries to access the Internet but is denied access, then it is told that the Internet is not currently available, in the same way it would be if the Internet really were not available. Similarly an app denied access to the contacts list is just told that the contacts list is empty. This means that the apps should cope gracefully with being denied access as this access denied responses are the same responses the apps would normally expect. A similar approach has now been deployed in Privacy Guard [136] on Cyanogenmod [253], a fully open-source fork of Android.

An orthogonal approach is to try and make the granting of the capability to perform some action part of the users normal interaction with the app (rather than a permissions dialog that interrupts their task). This could be done by using special UI elements that when pressed grant the permission and perform the action, such as pressing a camera icon allowing the app to take a photo [204]. However implementing this in practice is hard as trusted UI components are difficult because they need to clearly look like the trusted UI components to both the user and the code that is ensuring that the component is displayed, while still fitting into the UI of the app.

Permission models on other platforms

Other mobile operating systems have taken different approaches to permissions. J2ME and Symbian, which predate Android and iOS, had more complicated models. Symbian was capability based with 20 capabilities (permissions) with signed and unsigned apps and permissions were only available to apps that had been signed by a third party [155]. It could be complex to program for and depending on the configuration could repeatedly ask the user for permission every time an app needed it. J2ME had a security policy which was configurable by the manufacturer and could grant permission automatically or after asking the user whether they wanted to grant blanket permission, permission for that session or permission for one use, permissions could be revoked. BlackBerry also has a complex permission system. Users can refuse to grant permissions but sometimes they cannot later change this decision and since BlackBerry supports Android apps, it is not possible to change or revoke the permissions granted to Android apps. Windows apps can ask for permissions at install time or run time or in some cases just notify the user that permission has been granted. iOS apps

are verified by Apple and by default have more access than Android apps, they only have to request permission for GPS, incoming network connections, phone calls, SMS and email. However an analysis of Android and iOS apps found that iOS apps made more use of security sensitive APIs than Android ones, possibly because they do not have to tell the user they are doing this [130]. On both Windows and iOS it is not possible to restrict Internet access. While Android has relatively fine grained permissions and it used to only grant them at install time, Android M will support asking for permissions at runtime for apps that opt in to this.

2.5.2 Users cannot use and rely on developer reputation

If users cannot understand the permissions granted to apps, perhaps users can rely on the reputation of the developers of the apps? One difficulty with this approach is that it is hard to know which app users are using because a full screen app can completely control the display and appear identical to another legitimate app. One approach to solve this is to provide indicators, similar to those use for EV TLS certs in web browsers [28]. Another problem is knowing whether the app was produced by the organisation claimed. While app-ids are supposed to reflect the DNS hierarchy, there is no checking or enforcement as self-signed keys are used for signing apps. If the signing keys were published in DNS (as with DKIM) or signed by Certificate Authorities (CAs) (as with HTTPS) then that might help [242]. However, app signing keys have been shared between developers and so trust assumptions based on the security of the signing keys may be flawed [147]. If cryptographic solutions do not work, perhaps reputation based on reviews of apps can be used by users to evaluate the whether a developer or app is trustworthy? Unfortunately there is an underground market in app reviews allowing developers to skew this measure [262].

2.6 Vulnerabilities

Updates are required because of vulnerabilities and so this section describes work that analyses vulnerabilities, the economics and incentives behind discovery and fixes, and techniques for and instances of vulnerability discovery.

2.6.1 Analysis

There are three techniques for determining what versions of software users are running: analysis of network traffic, on device testing or historical analysis. These can then be used to determine whether the software is vulnerable. Network traffic has been used by analysing User-Agent HTTP headers to determine which web browser versions were in use [106]. On device testing has used databases of which files are on disks and their checksums such as the WINE database, which was used to detect the installed version of programs [177]. Historical analysis of published information was used on individual Android models to determine what versions were available for them over time [102].

When information on which versions are running on which devices is available, this then needs to be tied to which vulnerabilities affected those versions. Vulnerabilities in Android have been classified [243] and Chapter 3 describes how I collected data on vulnerabilities in Android and the versions of Android affected by those vulnerabilities. Existing vulnerability databases such as NVD can also be used [107], but may need to be cleaned before use [177] and may be biased [165]. By examining the NVD database, the estimated average number of zero-day vulnerabilities in existence is 2 500–4 500 [167]. Analysis of the exploitation of zero-day vulnerabilities has found that the typical attack lasts for 312 days (i.e. it is, on average, 312 days from the start of the attack until the vulnerability is publicly disclosed) and that the volume of attacks increases after disclosure by up to 5 orders of magnitude [29]. Early exploit incident reports can be used to predict the severity of a vulnerability and hence the priority of fixing it [36].

When these data are available then various metrics can be computed to try and evaluate the vulnerability of different platforms. §6.2 proposes a new metric and compares it with existing metrics.

2.6.2 Economics

The vulnerability of systems is strongly influenced by economic incentives. Microsoft put substantial effort into improving its security when the repeated discovery of vulnerabilities in its IIS stack damaged its reputation so much that users were advised to move to other platforms [188]. It is hard for potential customers

to evaluate the security of software (often vendors do not disclose vulnerabilities [51]) which makes software security a lemons market. One solution is to run auctions for vulnerabilities that allows companies to publish how much they will pay for the discovery of vulnerabilities in their systems [183]. It has been argued that if vulnerabilities are stochastically distributed then there is no point looking for vulnerabilities or reporting ones that are discovered (only in reporting ones that are used) [201]. However investigations of OpenBSD, found that vulnerabilities were not stochastically distributed and that older code that had already had vulnerabilities found in it was less likely to have new vulnerabilities found in it than newer code [184].

The policy on whether and when vulnerabilities should be publicly disclosed, that organisations such as CERT use, influences how quickly and if vendors fix vulnerabilities. Since the vendor does not internalise the full cost to its customers of vulnerabilities, it might release fixes later than is socially optimal. However the threat of disclosure can make the vendor release fixes more promptly [17].

2.6.3 Discovery

How are vulnerabilities discovered? There are various techniques from accidental observation and manual inspection through to fully automated strategies. While releasing and deploying updates improves security, exploits can be generated automatically from patches [39] and publishing the source code of the fix leaks the vulnerability [20], which is particularly problematic for open-source projects.

Automated techniques can find many vulnerabilities. The Woodpecker tool found permission leaks in stock phone images [126], automating a manual approach [80], and ADDICTED compared manufacturers customised images with stock AOSP images to find device file permission vulnerabilities [270]. Large scale analysis of firmware that looks for common vulnerabilities across many released binaries can find many instances of the same class of vulnerability [58], and static analysis of source code can also be fruitful. TLS implementations are difficult to get right and so static analysis of the OpenSSL source code has been used to find vulnerabilities [152] and analysing the composite state machines of TLS has revealed flaws in many implementations [27].

Analysing APIs can reveal flaws, such as attacks on iOS where apps could

steal other apps' secrets by adding themselves to the ACLs [263]. Similarly, HTML5 apps can have code injection flaws if strings are improperly escaped [141] and hybrid apps can break the same origin policy that is enforced in web browsers [115]. However, there has also been an attempt to strengthen the same origin policy by automatically wrapping websites in native apps [176] so hybrid apps do not always decrease security.

2.7 Updates

Since flaws will be found in existing systems and new techniques that better protect users will be developed, mechanisms for supplying updates are required and updates need to be supplied. CESG, which advises the UK government on computer security, advises that phones should be bought from manufacturers that supply updates [46] but, as shown in Chapter 6, manufacturers often do not. There have been legal efforts to force manufacturers to ship updates, particularly through cases brought to the FTC by ACLU [215].

2.7.1 Methods

The incentives for participants in the software distribution system are not always well aligned, which can result in software installation mechanisms that do not protect the users [7]. Often, software updates are not shipped in a secure way [22] or the package manager itself can be attacked [44]. Privilege escalation attacks have been found that exploit the update process for Android, allowing apps to be granted new permissions without asking the user [264]. Since manufacturers often do not ship updates for Android, PatchDroid tries to independently supply updates [172].

An analysis of the Windows XP update mechanism found that when the default was changed so that updates were installed automatically by default, less than 10% of the computers contacting the update servers did not have all updates installed [116]. In web browsers, analysis of User-Agent HTTP headers has shown that silent updates (i.e. the update is installed without asking the user) boost uptake and allow for rapid deployment of updates [86].

2.8 Mobile malware detection

Secure apps[2] rely on trusted devices and mobile phones are frequently used as that trusted device [34]. Since there are vulnerabilities in the platform strategies for coping with this fact are needed. Malware (malicious software) is software with a malicious purpose, often malware will exploit vulnerabilities in software or user understanding to achieve its purposes. Malware cannot exploit vulnerabilities if it can be detected and blocked before it runs.

Mobile security is an interesting area [92] as mobiles have capabilities that ordinary desktops lack (send premium SMS, use GPS), feature better sandboxing of apps and have basic package management (although still behind the state of the art found in Debian in 1996). These features could alleviate some of the problems with downloading and installing untrustworthy executables found on the Internet, and in the case of Google Play, this appears to be fairly successful [124].

Currently the main mechanism for compromising mobiles is malicious apps. Schmidt et al. predicted that Android would be the next target for malware shortly after it was released [207] and proposed an initial detection mechanism based on static analysis of function calls in ELF files [208]. Subsequently, there has been lots of work on analysing and detecting malicious apps.

To detect malware it is first necessary to determine that an app is behaving in a malicious manner. One way in which apps can behave in a malicious manner is to exfiltrate private user data such as location or contact details to the malware controller. TaintDroid [94] uses taint tracking to trace the flow of such data through the app and detect whether it is sent over the network interface. This is useful for security researchers analysing apps but unfortunately taint tracking cannot be used to protect end-users as it is hard to prevent the transmission of data through covert channels. Data leaks can also be detected using VetDroid's approach of tracking how permissions are used, which finds more leaks than TaintDroid [269]. Another approach is to use crowd sourcing on many end-user devices to obtain and detect malware by collating system call traces for apps and detecting malware as anomalies [42]. Alternatively, the Airmid approach is to monitor and detect network traffic that is indicative of malicious behaviour and

[2]Apps used for security critical tasks.

then use protected software on the phone to work out the origin of the traffic and take appropriate action [175].

To detect malware it is necessary to know what existing malware looks like, often by statically analysing the decompiled source code using tools like `ded` [93]. PlayDrone collected 1 100 000 hopefully benign apps from Google Play, which provides a good training set of benign apps [244]. The Android malware genome project makes a large set of >1 200 malware samples from 2012 available to researchers [271] (unfortunately at time of writing this was quite dated). It showed that the best malware detection rate by security software was 80%, 37% contained root exploits (the rest just ask for the permissions required) and that 86% of malware repackages existing legitimate apps. Detecting repackaging is one way of detecting malware, which is used by AnDarwin [59]. Most Android malware is distributed through alternative markets that are less well regulated [272]. Some malware has been detected through the characterisation of the permissions required by malicious apps' current monetisation strategies (such as sending premium rate SMSes) though careful filtering is required to get the number of false positives down to an acceptable level [272]. Similarly, RiskRanker searches for apps that appear to have risky behaviour such as running root exploits or having permissions that allow them to spend the user's money [125].

One difficulty with analysing Android malware is tracking control and data flow across different components, running as different Linux users and in different processes [91]; this is solved by the Amandroid framework [252].

However, the problem of detecting malicious apps is a hard one, as is defining malicious. Fundamentally determining whether a given app is malicious is reducible to the halting problem [55]. In practice, efforts by anti-virus apps on Android to detect malicious apps are ineffective, partly because there is no way for them to gain greater power than malware or prevent malware from being installed. DroidChameleon used simple automated transformation techniques on known malware samples and was able to fool all the mobile anti-malware products tested [197]. There have been attempts to make malware detection more robust against obfuscation, using semantic-aware analysis [268], or dependency graphs [156], but it is hard to evaluate their effectiveness against real malware. Real malware can try to bypass these techniques, which may also have worse performance (both in terms of compute time and in terms of false positive/false negative rate) than simpler analyses, such as Drebin, which does lightweight ma-

chine learning based on strings in the app and is easier to obfuscate [18]. It is also a little harder for anti-malware products to work on Android than on Windows because Android lacks APIs for anti-malware products to use and so it is harder for them to intercept installation or to do dynamic analysis on running apps. This is likely because Google intends that no malware gets through to the users through the Play Store because Google analyse the apps, which Google have found to be mostly true with only 0.15% of devices that only used the Play store, had a 'potentially harmful' app installed [124]. Google's Bouncer tool for scanning apps is possibly similar to AASandbox which combines static analysis of the binary with dynamic analysis in an emulator [31]. This does not work for third party markets that are the main sources of malware. However, Google have also deployed Verify Apps, which runs on Android devices to detect malware that arrives via other routes.

2.9 Mitigation in the face of malicious apps

Since detecting malware is hard there are efforts to mitigate the threats posed by undetected malicious apps. Three approaches are:

1. Compartmentalisation and containment – improving the sandboxing of all apps.
2. Helping developers to write secure apps.
3. Data eviction – reducing the quantity of data on the device that could be compromised.

2.9.1 Compartmentalisation and containment

One approach to mitigate the threat of malicious apps is to improve the compartmentalisation and containment used so that malicious apps cannot actually do anything dangerous even when installed and running on the device. Android has a permission system and apps have to request permissions on installation to use them later e.g. to access the GPS or camera.

Unfortunately, the permission system has holes in it, even without requesting any permissions on many devices it is possible to confuse other apps into exercising the permissions the other apps have on behalf of a malicious app [126]. For

example, an app with no permissions could wipe user data, send SMS messages and record the user's conversation.

Apps can be further sandboxed with additional technologies beyond those already deployed in Android. There are four levels at which compartmentalisation has been attempted, verifying the code before it is executed, wrapping the code inside the address space of the process, interposing system calls in the kernel or using a hypervisor.

Aurasium is a tool that repackages Android apps to intercept system calls inside the address space of the process and so provide additional checks that the user could define such as checking for premium rate SMSes or access to particular blacklisted IP addresses [266]. One limitation of this work is that it relies on the correct operation of the Dalvik VM for its security and the Dalvik VM is not inside the TCB and has not been written to be secure (unlike the standard Java Virtual Machine (JVM), which was written to be secure and yet is very insecure).[3] Hence, while this technique is useful for wrapping unsuspecting apps, it does not stop a determined attacker who knows about this technique. It also does not wrap native code, a problem addressed by AppCage, which uses software fault isolation to provide a native sandbox [273].

A similar approach in the browser space is Native Client [267], which runs native x86 code securely inside the browser by forcing a particular structure on the code that makes it tractable and efficient to verify that it does not have any malicious behaviour as all control flows can be safely overestimated statically. This requires the code to have been compiled originally with a non-standard tool chain but does allow C/C++ code to be ported to run in browsers without compiling to JavaScript.

Another layer in the Android stack at which additional compartmentalisation can be enforced is the Linux kernel: SEAndroid [213], which brings SELinux to Android and so enforces Mandatory Access Control. This protection stops many known exploits and stops much misbehaviour by apps, but it does also forbid some valid behaviours and so may not be completely compatible with all existing apps. Instead of forbidding access to resources to prevent vulnerabilities being exploitable, the potentially malicious code can be interfered with in other ways that do not compromise correctness of operation, but may stop exploits

[3] This insight comes from David Chisnall.

from working, such as slowing the code down if it executes suspicious system calls [134].

A hypervisor could be used below the kernel to provide additional compartmentalisation. Cells uses a hypervisor to provide multiple virtual phones running on one device [14]. This allows apps for work and personal use to be on different virtual phones and so not interfere with each other. Another hypervisor approach is L4Android, which uses the L4 microkernel hypervisor [150]. A lighter weight approach is to perform namespace/filesystem isolation only for untrusted apps, such as that used by AirBag [261], but this might not have all the security benefits of a hypervisor.

Some policies are context dependant and CRePE allows policies on what apps are allowed to do to vary with context [57]. Even if a malicious app were to get permission to use certain sensors, all apps could be denied access to them in certain situations. For example, the camera and location information could be disabled when working in a secure facility. PEDAL takes a different approach to context and determines the context of calls into Android platform code in terms of which code it came from, in particular whether the call came from an ad-library and enforces different policies accordingly [157].

2.9.2 Writing secure apps

Another way of reducing the attack surface is to make it easier for developers to write secure apps. For example, Capsicum [251] allows developers to sandbox parts of their own app so that the sandboxed code only has access to certain specified file descriptors and can gain no additional access to anything not given to it. This means that, for example, exploiting the decompression library only allows the attacker to change what ends up in the decompressed file and the attacker could already do that by supplying a different file to decompress.

2.9.3 Data eviction

Even with all these strategies to prevent or limit compromise there is still the risk of an attacker succeeding in gaining control of the device, either through an exploit or through gaining physical access. One solution to mitigate that threat is to limit the amount of data on the device.

On flash storage, due to the need to erase large blocks and the limited number of erase cycles each block can support, wear leveling makes deletion more difficult, and so there is more likely to be residual data on the flash storage than with magnetic hard disks. The data node encrypted file system encrypts all the files on disk and uses careful management of the keys so that it can forget the key used to decrypt a particular file and hence permanently delete it [200].

CleanOS solves this differently by trusting a cloud service to securely store the keys used to decrypt the data stored on the device and evicting the keys used from the device whenever the data are not in use [228]. This places a lot of trust in the cloud service, but since the data was probably already stored in plaintext in the cloud, this might not increase the trust already given to the cloud provider. However, it does also require continuous Internet connectivity, which is not available in practice.

2.10 Summary

Users face substantial threats. Security-minded developers would like to build secure software to protect them, but doing so is hard, particularly given the difficulty users have in understanding what is happening and the risks involved. Progress has been made in improving user understanding and providing better information and mechanisms. However, security depends not only on the features of the platform, but also on the vulnerabilities in the platform, which allow behaviours that are unauthorised. These vulnerabilities allow malware to perform unauthorised actions, the response to this is to try and detect malware and to mitigate the damage it can do, mostly through compartmentalisation and containment.

This chapter has identified a hole in understanding that this dissertation will shrink. Information on what vulnerabilities affected Android was not available (Chapter 3) and neither was information on how this affected the security of Android as a whole (Chapter 4) or how the deployment of updates to Android could be modelled (Chapter 5). Lack of quantifiable comparative information on the security of Android creates a lemons market for Android manufacturers as no one can tell who provides better security (Chapter 6).

CHAPTER 3

CRITICAL VULNERABILITIES IN ANDROID

Support for third-party apps is a key feature of modern smartphone operating systems. Such apps are written by many developers from a wide range of backgrounds, which means neither operating system vendors nor users should fully trust app developers. Therefore, in order to secure personal data and prevent theft, such as the sending of premium-rate text messages or exfiltrating user data, operating system vendors provide a protected execution environment, or *sandbox*, for apps.

Not all malicious apps need to break the sandbox in order to misbehave, many instead just ask for the permissions required and rely on the user not checking them [103]. Here the focus is on the issue of apps breaking out of the sandbox because malware which is able to take full control of a handset can do significantly more harm and is much harder to remove. Ordinary malware can be uninstalled by the user, or remotely by Google through the Play Store, and returns the device to a secure state. Removing an app which has used an exploit to take control of the handset is unlikely to return the device to secure state.

Using Android for security critical tasks relies on the platform being secure, but definitive information on whether it is secure is hard to find. This chapter describes the threat model and three attack vectors (§3.1) and explains how I created AndroidVulnerabilities.org (AVO) [232], a website for recording information on critical vulnerabilities and describe the vulnerabilities recorded. In the next chapter these data will be used to compute the exposure of Android to critical vulnerabilities.

Some of the material in this chapter was combined with material in other chapters and accepted for publication at the 5th Annual ACM CCS Workshop

on Security and Privacy in Smartphones and Mobile Devices (SPSM) [236]. The ideas for the threat model and of critical vulnerabilities arose from discussions with Alastair R. Beresford but the data analysis and presentation are my own work.

3.1 Defence and attack vectors: Threat model

There are three levels or rings of defence at which a defender can deploy security controls: in an online marketplace, at app installation, and during app execution.

1. The first level of defence comes from an app marketplace such as Google Play. A marketplace can use three mechanisms to prevent malicious apps from appearing in the marketplace: firstly by economic disincentives, including charging developers to create an account and cancelling malicious accounts; secondly by technical means, such as static and dynamic analysis on submitted apps (e.g. Google's Bouncer); and finally via social feedback from users, including the provision of reviews, rating and reporting.

2. The second level of defence occurs on the handset at installation time, and includes checking the signature of any app installation or update as well as checking app for malware (e.g. Google's Verify Apps feature).

3. Finally, the third line of defence takes place at runtime: Android apps run in a *sandbox* that prevents the app from unauthorised actions such as accessing any permissions which weren't granted by the user at install time. In addition, anti-malware products may scan the installed apps periodically to detect malware.

There are also three attack vectors against Android handsets that need to be protected against: the installation, dynamic code loading, and injection attack vectors.

- The *installation* attack vector affects the process of installing a malicious app on the device. Android devices can install apps through marketplaces, email attachments, URLs and via the Android Debug Bridge (ADB). By default, many Android devices disable installation of apps from other sources

and will only allow the installation of apps from Google Play, which uses Bouncer (a level 1 defence) to automatically analyse apps, and quickly takes down apps that are reported as malicious. However, alternative markets are also popular, particularly in countries where Google Play is not available. In 2012 0.02% of apps on Google Play and between 0.20% and 0.47% of apps on alternative markets were malicious [272]. Google estimates that, in 2014, fewer than 0.15% of devices which only install apps from the Google Play Store have Potentially Harmful Apps installed [124].

- The *dynamic code loading* attack vector occurs when an app downloads and executes code at runtime. The most direct attack strategy is to upload to a marketplace a seemingly innocent app that dynamically loads malicious code, either as additional Dalvik bytecode, as a native library or by embedding an interpreter and executing received instructions. Neither static nor dynamic analysis of this app (level 1) will uncover any malicious code, since it does not exist in the app. The marketplace can try to detect explicit use of dynamic code loading, however there are ways to dynamically load code which are hard to detect, even on a platform such as iOS, which does not permit dynamic code loading. For example, a Return-Oriented Programming (ROP) attack is relatively easy if the attacker creates an app with carefully crafted flaws [249].

- The *injection* attack vector occurs when the attacker injects malicious code directly into existing code already on the handset. For example, the addJavascriptInterface vulnerability (CVE-2012-6636), detailed in Chapter 5 allows JavaScript running in an Android WebView to execute arbitrary code as the vulnerable app's user. The fix for the addJavascriptInterface vulnerability breaks backwards compatibility and requires a two-sided fix.

The best place to prevent attacks is at runtime, since all three attack vectors can be prevented at this level. Unfortunately, as shown in the next chapter the sandbox for Android apps is weak due to the persistent presence of known vulnerabilities. Therefore many Android users implicitly rely on the marketplace for protection by removing apps that are detected as malicious. This detection can only be derived from static and dynamic analysis or after reports of malicious behaviour from users once it is made available for download.

3.2 Critical vulnerabilities

To evaluate the exposure of Android devices to these threats, information on the vulnerabilities in Android is required. I compiled a list of critical vulnerabilities in Android, containing information on the discovery and publication dates, the versions affected and which versions fixed the problem. Only critical vulnerabilities, such as root vulnerabilities that did not require USB debugging to exploit, are included. Critical vulnerabilities allow a program to gain privileges equivalent in scope to root. If an app exploits a critical vulnerability then it gains control of the device. Some phones can be 'rooted' by enabling USB debugging and using the special privileges of the ADB shell to root the device but only 19.4% of devices in the Device Analyzer dataset[1] have USB debugging enabled. This is not something that apps running on the phone can exploit to break out of the app sandbox and so those vulnerabilities are not included. Unfortunately, many published exploits use ADB for convenience and so determining whether the use of ADB is necessary to exploit the vulnerability can be difficult.

Some critical vulnerabilities are not traditional kernel vulnerabilities, for example the discovery of flaws in the verification of signatures on Android apps in February 2013 [105] meant that apps could pretend to be signed with system keys and hence gain root equivalent privileges. On some versions of Android (below version 4.1) malware could use known system-to-root escalation mechanisms but on all versions the apps have a greatly increased attack area for further privilege escalation and also have the ability to control all user Internet traffic via VPNs, brick the phone, remove and install apps, steal user credentials, read the screen and make as well as receive calls. This is shown in the categories used in Table 3.1, which distinguishes between 'signature', 'system' and 'kernel' vulnerabilities.

[1] Discussed in §2.4 and §4.1.

Vulnerability	How known	Date	Categories	CVEs
KillingInTheNameOf	Fixed on	2010-07-13	system, kernel	CVE-2011-1149 [61]
exploid udev	Discovered on	2010-07-15	kernel	CVE-2009-1185 [60]
levitator	Discovered on	2011-03-10	kernel	CVE-2011-1350 [62], CVE-2011-1352 [63]
Gingerbreak	Fixed on	2011-04-18	system	CVE-2011-1823 [64]
zergRush	Discovered on	2011-10-06	system	CVE-2011-3874 [65]
APK duplicate file	Discovered on	2013-02-18	signature	ANDROID-8219321, CVE-2013-4787 [66]
APK unchecked name	Discovered on	2013-06-30	signature	ANDROID-9950697
APK unsigned shorts	Fixed on	2013-07-03	signature	ANDROID-9695860
Fake ID	Fixed on	2014-04-17	signature	
TowelRoot	Discovered on	2014-05-03	kernel	CVE-2014-3153 [67]
ObjectInputStream	Discovered on	2014-06-22	system	CVE-2014-7911 [68]
Stagefright	Fixed on	2015-04-08	system, network	CVE-2015-1538 [69], CVE-2015-1539 [70], CVE-2015-3824 [71], CVE-2015-3826 [73], CVE-2015-3827 [74], CVE-2015-3828 [75], CVE-2015-3829 [76]
One class to rule	Discovered on	2015-05-22	system	CVE-2015-3837 [77], CVE-2015-3825 [72], ANDROID-21437603, ANDROID-21583849
Stagefright2	Discovered on	2015-08-15	system, network	CVE-2015-6602 [79], CVE-2015-3876 [78]

Table 3.1: Critical vulnerabilities in Android

In this dissertation critical vulnerabilities are defined as vulnerabilities that allow for complete control of the device without requiring special access, such as physical access or code signed with a special key.

The AndroidVulnerabilities.org (AVO) [232] website is an open platform for filing critical vulnerabilities in a machine readable format. It was seeded it with data from the CVE database, vendor lists, reports from the literature and various forums. In addition, it received submissions or amendments from 11 individuals. Data was collected between 2013-08-28 and 2016-03-21 and so any information lost before the start of that period cannot be included. AVO currently contains 42 vulnerabilities of which 20 affect all Android devices and 22 are specific to particular devices or device manufacturers.

The 14 vulnerabilities that fit the attack vectors introduced in §3.1 are used in this analysis, and are shown in Table 3.1. These vulnerabilities affect all Android devices regardless of manufacturer, and as a result the selected vulnerabilities will dominate security analysis of the Android ecosystem as a whole. In many cases manufacturer- and model-specific vulnerabilities cannot be matched to records from other data sources and therefore attempting to include device-specific vulnerabilities as well would introduce additional uncertainty into the results. In contrast, with this set of vulnerabilities, this analysis represents a lower-bound on the vulnerability of devices.

Tracking vulnerabilities is a manual task as they were not consistently recorded in other databases such as the CVE database. In addition, the lack of a widely acknowledged unique identifier required manual analysis to identify whether two reports referenced the same vulnerability. Previous work has assumed "any security issue of relevance will eventually get a CVE number assigned" [108]. This is currently not the case for critical Android vulnerabilities. For some of the vulnerabilities without CVE numbers, a Google engineer confirmed by email that there was no CVE number and that they did not intend to get one, instead providing an internal Android bug number.

3.3 Lifetime of a vulnerability

The key events in the lifetime of a vulnerability do not always occur in the same order and are:

creation When a vulnerability was created in the source code.

introducing release When the first release was made containing the vulnerability.

discovery When the vulnerability is first discovered.

exploit When the vulnerability is first exploited.

disclosure When the vulnerability is first disclosed.

fix When a vulnerability was first fixed in the source code.

fixing release When the first release containing the fix was made (equivalent to 'patch available' [108]).

fix deployed When the fix has been deployed to a sufficiently large proportion of the population that the vulnerability can be ignored (the ecosystem equivalent to the per instance 'patch installed' [108]).

Establishing when a vulnerability starts to pose a threat to users is difficult. Frei et al. [108] propose the definition of the **time of disclosure**. This occurs when the information about the vulnerability is freely available to the public from a widely accepted and independent source and has been validated by security experts so that it has a risk rating. Unfortunately before I collated this information, much of it was not published by an independent source and lacked risk rating information, even months or years after the vulnerability had been actively used. Therefore this measure does not work.

Symantec's 2012 analysis of desktop malware has shown that after public disclosure, exploitation rates increase by 5 orders of magnitude [29] and so from the point of view of widespread danger, the period between public **disclosure** and the date of **fix deployed** is the most critical. However Symantec's analysis also showed that zero-day vulnerabilities were typically used for 312 days before they were publicly disclosed, often to target particular organisations. Hence when considering a vulnerability from the point of view of an organisation that cares about Advanced Persistent Threats, such as those responding to CESG advice [46], the critical period starts with **discovery** and continues until **fix deployed**. Therefore, in the vulnerability calculations the earliest recorded date of **discovery** is used, even if that knowledge might have been confined to a particular device manufacturer or hobbyist.

Unfortunately **discovery** is the hardest point to obtain concrete data on as discovery may happen multiple times independently and not all discoverers will report their discovery. **Creation** and **introducing release** are the earliest points that could be used, but the risk is mostly latent until someone discovers them[2] and therefore these dates are not used. The date of first **exploit** is a point when the risk is definitely high, but a competent adversary may not be detected when using such exploits. The date of first **fix** is, assuming that the fix is deliberate, a point at which the vulnerability is known at least within the organisation performing the fix and frequently implies an earlier discovery and notification by a third party; the date is determined from the Android Open Source Project repository by examining the authored-on date of the fixing commit. Once a **fixing release** has been made then the vulnerability is widely known because it can be reverse engineered from the changes in the release [39]. When the fix is deployed to a sufficiently large proportion of devices then the remaining risk is minimal. Therefore the earliest known date from **discovery**, **exploit**, **disclosure** or **fix** is used. A breakdown of the type of date used per vulnerability is shown in Table 3.1.

3.4 Distribution of vulnerabilities

Figure 3.1 shows the dates of **discovery** and, when later, the date of the first **fixing release**. Some vulnerabilities (*levitator*, *zergRush*) were fixed in released versions of Android before they were discovered and so are shown as vertical lines, while others were known for months before a version of Android that fixed them was shipped. During the period in which Device Analyzer data was collected the date when a version of Android with the fix was observed on a Device Analyzer device is taken as the date that version was released. For vulnerabilities prior to the collection period I estimate the release date as best I can using publicly available data. There is no canonical source of Android release dates, and my best guesses and supporting references are available from AVO [230].

The discovery dates of these vulnerabilities does not appear to be uniform. In particular the data shows a large gap from 2011-10-06 to 2013-02-18, with no recorded discoveries of critical vulnerabilities affecting all Android devices.

[2]This is not the case for intentionally introduced vulnerabilities/backdoors as they are known at creation, but none of the vulnerabilities in AVO are suspected of being intentional.

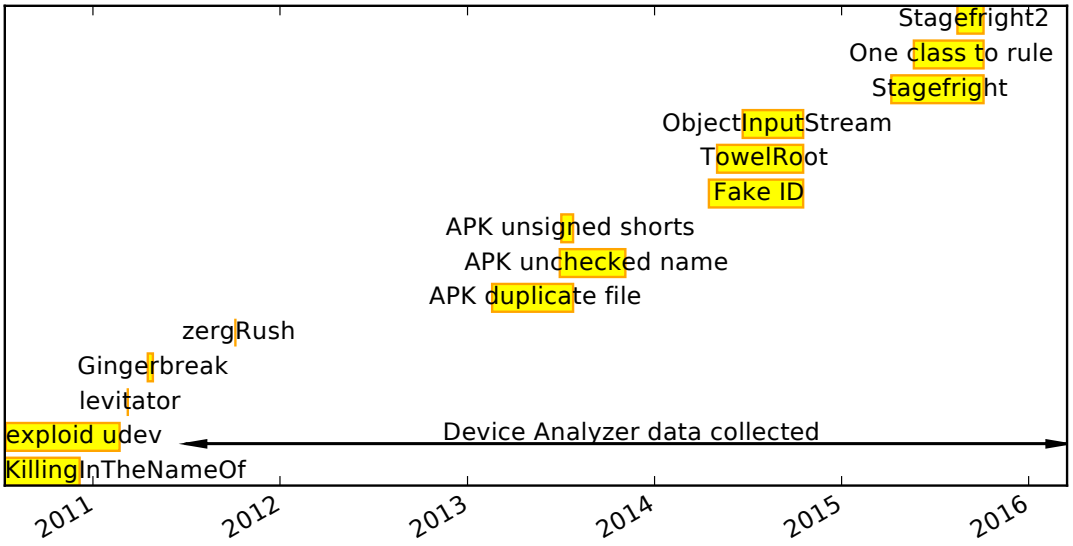


Figure 3.1: Timeline of vulnerabilities. For each vulnerability this shows the dates of **discovery** and, when later, the dates of the first **fixing release**.

The cause of this quiet period is unclear. Possible explanations are that: (i) most devices were exposed to known vulnerabilities so there was no point in looking for new ones (this is shown later in Figure 4.1); (ii) device manufacturers made it easier to install custom versions of Android, reducing the need for users to root their devices; and (iii) device manufacturer specific vulnerabilities (which are not included in this analysis) were easier to find and therefore attackers looking for vulnerabilities adjusted their focus. Due in part to the development of Android 3 and the fact that it was only used for tablets, Android 4 did not come to dominate the versions of Android in use until 2013 when new vulnerabilities were discovered.

3.5 The selection of vulnerabilities in AVO

Later analysis does not include all the vulnerabilities from AVO. In particular, it does not include any device manufacturer specific vulnerabilities, even when these are widespread (such as vulnerabilities in Qualcomm[3] chipsets) because it is difficult to work out which devices are affected. There also tends to be less

[3]Qualcomm is particularly good at publicly disclosing vulnerabilities and the patches that fix them [54].

public information available about manufacturer-specific vulnerabilities, which makes them harder to tie down.

For some other vulnerabilities that might affect all Android devices it is hard to work out which devices are affected. For example, *pty race* is a Linux kernel vulnerability. I found 48 commits that fixed this vulnerability on different branches. On some branches it was also accidentally fixed, and later reintroduced, before finally being fixed properly, and therefore determining which devices were vulnerable relies on knowing where, and on which branch, the kernel code for a specific release was taken from.

For some vulnerabilities the required patch is not in AOSP. This is despite the fact that some device manufacturers have shipped builds containing the fix. This makes it difficult to determine whether a specific device is vulnerable. For *RageAgainstTheCage adb* and *keystore buffer* vulnerabilities, there is insufficient data to determine whether they fit one of the attack vectors as the former may require physical ADB access and the latter may be thwarted by the sandbox. However a sensitivity analysis showed that those two vulnerabilities make little difference to the results presented later.

3.6 Summary

This chapter described the threat model and attack vectors used later, collected together existing ideas and detailed the creation of a website and database to record information about critical vulnerabilities in Android. It summarised information about these vulnerabilities and explained the difficulties involved in collecting concrete data on them and evaluating the impact these vulnerabilities have, particularly since, contrary to prior assumptions, not all critical vulnerabilities have CVE numbers assigned. This shows that there is a steady supply of critical vulnerabilities affecting Android, hence a need to quickly deploy security fixes to devices. The next chapter investigates whether this happens.

CHAPTER 4

ONLY THE STORE CAN SAVE YOU NOW: UPDATES AND VULNERABILITIES ON ANDROID

Modern mobile operating systems combine up to three layers of security protection to prevent app malware from taking control of a device. These are: the defences provided by the app store, additional checks performed at installation time on the device, and a protected execution environment to control the running app. This chapter shows that a number of well-known, publicly disclosed vulnerabilities (detailed in the previous chapter) continue to allow apps to bypass installation checks or bypass runtime restrictions. This leaves the app store and Verify Apps/anti-virus apps to provide protection for the majority of Android users. It quantifies for the first time the vulnerability of Android devices in the wild and how long it takes for critical flaws to be fixed on consumer devices. This is the result of analysing the Device Analyzer data from over 4 years and 24 600 devices and found that, on average, 88% of devices were exposed to known privilege-escalation attacks. Furthermore, devices apply 1.4 updates each year, less than the critical vulnerability discovery rate of between 3.8 and 8.0 per year.

In this work, critical vulnerabilities are those that allow malware (in apps (malicious or compromised), or content such as webpages or videos) to gain full control of the device rather than merely abuse permissions granted to it by the user. The reason for this focus is that the negative impact on the device owner is much higher for malware that gains root than malware that does not. This is because it may not be possible to restore a device that has been affected by malware that has gained root to a clean state and it may perform any action on

the device without the restrictions that would affect other malware.

Sandboxes only provide protection if they are free from design and implementation flaws. Both design and implementation flaws have occurred in Android, as indeed happens today in many large security systems relying on the sandbox model. Maintaining sandbox security therefore relies on providing timely updates to fix design and implementation flaws as and when they are discovered.

This chapter shows that the Android sandbox only provides effective security on 12% of devices – in contrast to Google's reported claim that 100% of Android devices are protected [185]. This is because most devices do not receive timely updates. Therefore the security of the Android ecosystem is currently reliant on the defences provided by the app marketplace [124]. This insight allows us to provide users with a list of concrete actions to improve device security. It also raises broader questions about how modern mobile device security is most effectively achieved, how a secure ecosystem should be constructed and managed, and whether security by obscurity is the right way forward.

Some of the material in this chapter was combined with material in other chapters and accepted for publication at the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) [236]. The data from Device Analyzer was collected using code written by Daniel T. Wagner and I used and extended frameworks he wrote to perform the analysis of this data presented in this chapter. I performed the analysis, produced the graphs and wrote the text, Alastair R. Beresford and Andrew Rice provided vital advice and discussion of the analysis and its presentation.

4.1 Data sources

Two sources of data are required to determine whether the Android sandbox is non-vulnerable or not: (1) information on the distribution of installed versions of Android over time and (2) information on the critical vulnerabilities found to affect specific versions of Android (discussed in the previous chapter). These two datasets can then be combined to determine the proportion of devices at risk of attack from specific vulnerabilities over time.

The uncertainty in results (the measurement error) is indicated by presenting

them $\pm$ one standard deviation and results are given to 3 s.f., this occasionally results in '± 0' when the standard deviation is small. Systematic errors are discussed in §4.3.1.

4.1.1 Versions of Android running on devices

This analysis uses historical data collected by the Device Analyzer project [246]. Device Analyzer collects data from study participants who install the Android app from the Google Play store. Most study participants allow researchers around the world to access a subset of their device data, including the data presented here.

The Device Analyzer app collects a range of data from Android devices [245]. I extracted the build string and API version for each device each day. The build string is a user-readable version string. The API version is a positive integer that increases when new features are added to the API. Consequently security (bug) fixes do not usually result in a change in the API version, the exception to this among the vulnerabilities mentioned in this dissertation is presented in Chapter 5. Fortunately most (99.9%) entries in these data have a build string of the form 'x.y.z opaque_marker' and so it is possible to extract the Android version number 'x.y.z'. On a large proportion of devices 'opaque_marker' is a well defined build number [13] however different device manufacturers use different schemata. Google provides API version distribution information but not the OS and build version information needed [231]. In §4.3.1 the API version data provided by Google is used verify that the Device Analyzer data is representative.

Device Analyzer has collected data from 24 600 devices and 1 520 000 device days in total. The majority of devices only contribute data for a short period of time, however 2 110 devices have contributed data for more than 6 months.

4.2 Analysis

This analysis shows that, on average between July 2011 and March 2016, $87.6 \pm 0.0\%$ of Android devices were exposed to critical vulnerabilities and only $5.67 \pm 0.0\%$ run the latest version of Android. Devices, on average, apply 1.43 ± 0.01 updates each year, which is less than the rate of critical vulnerability discovery of between 3.79 ± 0.84 and 7.96 ± 1.23. Of those version changes, $2.49 \pm 0.20\%$ are

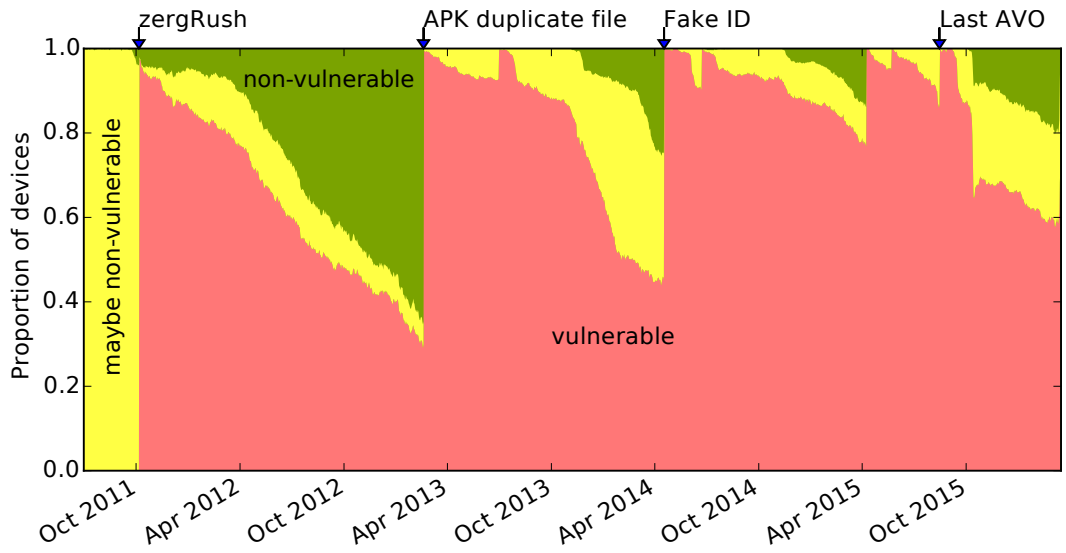


Figure 4.1: Proportion of devices running vulnerable, maybe non-vulnerable and non-vulnerable versions of Android against time. The red vertical cliffs are caused by the discovery of vulnerabilities, with those that have the greatest impact annotated. 'Last AVO' indicates the date of the last vulnerability included from AVO, data after this date overestimates security.

downgrades to older versions. This section describes three analyses, examining the vulnerability of Android devices in general (§4.2.1), exploring the upgrade cycle and vulnerability cycle of the ecosystem (§4.2.2) and quantifying the updates installed on devices (§4.2.3).

4.2.1 Analysis 1: Vulnerability of Android devices

In this analysis the proportion of Android devices that are exposed to critical vulnerabilities and how that has changed over time is calculated. Figure 4.1 summarises the proportion of Android devices susceptible to at least one critical vulnerability.

The OS version information from Device Analyzer and the vulnerability data from AndroidVulnerabilities.org (AVO) (Chapter 3) was used to investigate the vulnerability of Android devices. The AVO data covers the period from 2010-07-13 to 2015-10-22. The Device Analyzer data was collected between 2011-07-01 and 2016-03-16. For each device the daily version data and any of the 14 vulnerabilities discussed in Chapter 3 that it was exposed to at that time was recorded. These totals were then normalised for each day by dividing through

by the total number of devices with version information on that day.

A device is *vulnerable* if it is running a vulnerable version of Android and the device has not received an update that might fix it; it is *maybe non-vulnerable* if it is running a vulnerable version but received an update that could have fixed the vulnerability if it contained a backported fix; and it is *non-vulnerable* if it is running a version with no known vulnerabilities. This can be used to plot Figure 4.1. Initially all devices are *maybe non-vulnerable* (yellow) since Device Analyzer does not have historical data prior to July 2011. This means it is not possible to distinguish between devices that are running a version of Android that is known to be vulnerable from one that may have received a backported fix. This demonstrates the importance of a longitudinal study: this type of analysis requires years of data. Once *zergRush* was discovered in October 2011 then most devices are recorded as *vulnerable* (red) in Figure 4.1 as they were known to be vulnerable. The remaining devices were already running a version of Android that fixed the *zergRush* vulnerability and are therefore marked as *non-vulnerable* (green). From October 2011 until the discovery of *APK duplicate file* in February 2013 the graph shows progressive improvement as devices are upgraded or replaced. This means more and more devices are marked as *non-vulnerable* because they are now running a version of Android with no known vulnerabilities, or marked as *maybe non-vulnerable* because they received an OS update that did not update to a known-good version of Android but that may still have included a backport of a fix, as the update was made available after the vulnerability was disclosed. From February 2013 onwards regular discovery of critical vulnerabilities ensures that most devices are vulnerable. Discarding devices classed as *maybe non-vulnerable*, on average $87.6 \pm 0.0\%$ of devices were classed as *vulnerable* and 12.4% classified as *non-vulnerable* between July 2011 and March 2016.

4.2.2 Analysis 2: Behaviour of the Android ecosystem

This analysis examines how the distribution of Android OS versions changes over time and the impact that has on how different vulnerabilities affect the security of Android. It shows that some vulnerabilities continue to have a substantial affect long after they have been fixed, and the importance of longitudinal studies for determining whether devices are exposed to a particular vulnerabilities.

As in Analysis 1, I used the version information for each device to calculate which critical vulnerabilities each device was susceptible to each day.

The proportion of devices in the Device Analyzer data running different versions of Android each day is shown in Figure 4.2.[1] The anomaly beginning in August 2014 is explained in §4.3.3 and does not have a substantial affect on the results. The figure shows how old versions are gradually replaced by new ones, and the long tail of devices that do not see updates to more recent versions. This gives a mean proportion of devices running the most recent version of Android since 2011 of $5.67 \pm 0.0\%$.

The vulnerabilities devices are exposed to are shown in Figure 4.3. For each vulnerability it shows the proportion of devices exposed to that vulnerability and how that changes over time. The variation of the proportion of devices affected by a vulnerability with time tells us how badly a particular vulnerability affected the Android platform. In July 2011 the *exploid* and *levitator* vulnerabilities both affect most Android devices. Slowly these are fixed as updates roll out and devices are replaced until in January 2013 a much smaller proportion of devices are affected by known vulnerabilities. However when in February 2013 the first APK signing vulnerability was found. It affected all previous versions of Android and even in October 2013 most devices (92.2%) were still vulnerable.

In 2013 three vulnerabilities were found in how Android verified the signatures on APKs. These are *installation* vulnerabilities in the threat model since they require a new app installation to occur. Figure 4.3 shows how the *APK signing vulnerabilities* affected all devices and took months to get fixed for any device. However what is perhaps more worrying is the long tail on the *Gingerbreak*, *levitator*, *exploid* and *zergRush* vulnerabilities, which are more dangerous root privilege escalation vulnerabilities (since they do not require new app installation) and affected a significant proportion of devices many years later.

4.2.3 Analysis 3: Updates to particular devices

Previous graphs summarise data across all the devices, however one of the advantages of the Device Analyzer data is that it provides visibility into what happens to individual devices over time. This allows us to determine whether newer

[1] The raw number of devices running different OS-versions each day is given in Figure 2.4 in §2.4.

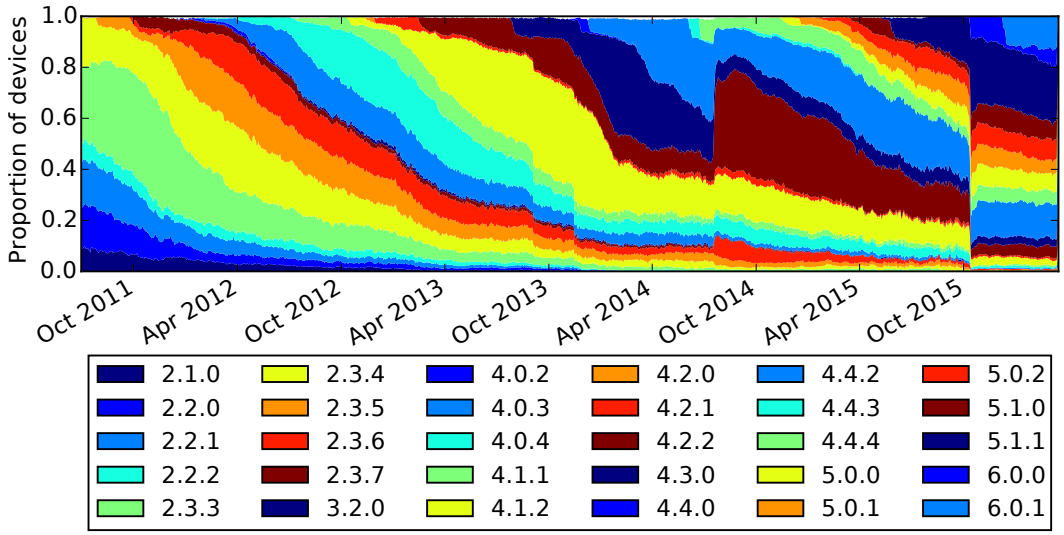


Figure 4.2: Android versions in Device Analyzer data over time. The change in behaviour after August 2014 is explained in §4.3.3.

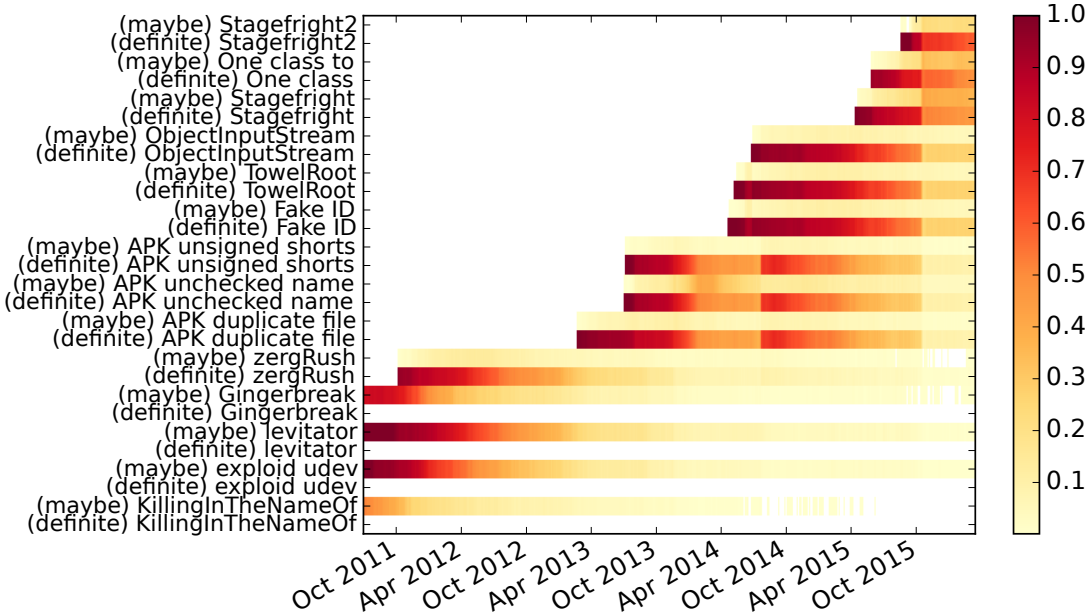
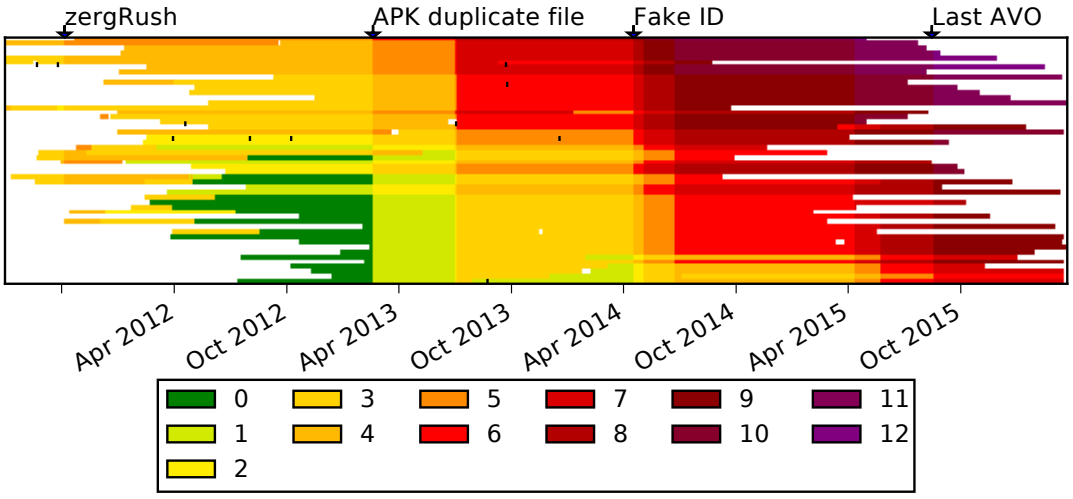
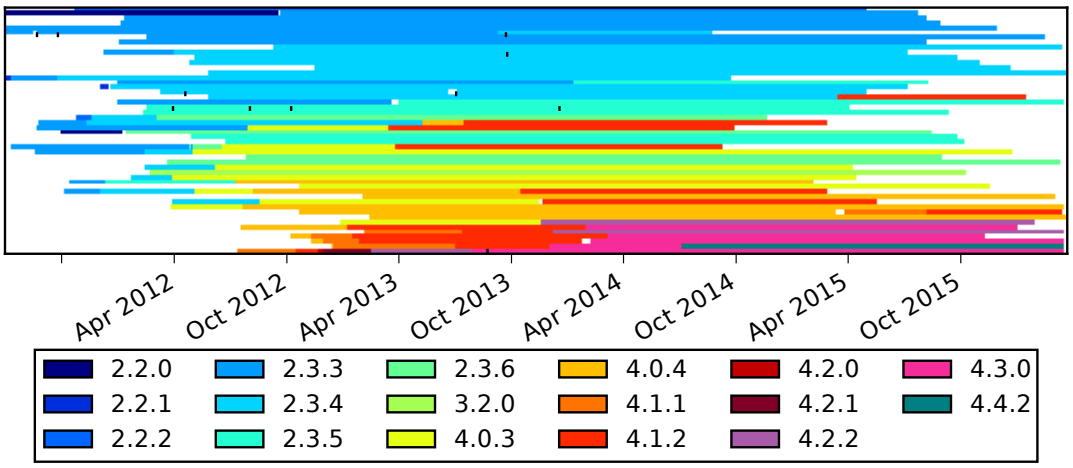


Figure 4.3: Proportion of devices affected by different vulnerabilities. The prefix '(maybe)' indicates 'maybe non-vulnerable' and '(definite)' indicates 'definitely vulnerable'. The first few vulnerabilities are all 'maybe non-vulnerable' as I do not have data from before that vulnerability was discovered while later ones are mostly 'definitely vulnerable' as I know no fixing update reached the devices. The change in behaviour after August 2014 is explained in §4.3.3. Individual plots for each vulnerability are given in §A.3.



(**a**) Each strip shows the number of vulnerabilities affecting each device over time.



(**b**) Each strip shows OS versions over time.

Figure 4.4: The top 50 devices by days of contribution in the Device Analyzer data. One strip per device handset. Black lines show where an update only changed the build number.

OS versions are used because people are buying new phones running newer OS versions or whether existing devices receive updates.

As in Analysis 1, the version information for each device was used to calculate which critical vulnerabilities each device was susceptible to each day. Changes in version were also recorded. In the Device Analyzer data there are 884 ± 341 devices contributing data in any particular week. However most devices only contribute for a short period of time and so updates are not observed on every device. Instead Device Analyzer provides a, hopefully representative, sample of updates that happened while Device Analyzer was installed on the de-

vices. Device Analyzer cannot distinguish between a device replacement and the removal of the Device Analyzer app as while multiple devices can be linked to the same user account, this is a manual process and few users do it.[2]

The longitudinal data on the number of vulnerabilities affecting the 50 devices that have contributed the most days of data to Device Analyzer changes over time and is shown in Figure 4.4a. Figure 4.1 showed a trend of security improving and then getting worse and this trend is also shown in Figure 4.4a. It shows how some devices had vulnerabilities, which were fixed, and then further vulnerabilities were discovered, and mostly not fixed. This implies that these devices had been abandoned by the device manufacturer and were not receiving updates. This is confirmed in Figure 4.4b, which shows which OS versions those devices were running. Some devices start off in 2011 exposed to known security vulnerabilities and are still exposed to those vulnerabilities, and additional ones, in 2014. The potential bias introduced by picking the top 50 devices by length of contribution is that these are devices that have been in use for a long time, perhaps longer than normal and hence might be more affected by manufacturers dropping support than the average user.

Device Analyzer recorded 5 970 update events and found most are upgrades (4 080). These are shown in Figure 4.5. While many upgrades are from one version to the next version there are also a fair number (891, 15.2%) that skip more than 3 versions. Surprisingly there are also a small number of downgrade events (146, 2.49%) when older versions of Android are installed on to devices. Possible reasons for downgrading include freeing up space on the device, to make it easier to root or because a new version introduced bugs or performance problems.

The number of devices that received security updates each week, is shown in Figure 4.6. Updates that changed the Android version number so that the number of known vulnerabilities decreased are shown in red. Updates that changed the build number but not the version number and so might contain a backported fix for a vulnerability are shown in yellow. Updates that did not fix security vulnerabilities (because there were no known security vulnerabilities in the existing version of Android) are shown in green.

By dividing the number of updates observed by the number of device years of

[2]In general there are other mechanisms that could be used, such as tracking the Google account used, but for privacy reasons Device Analyzer does not do this.

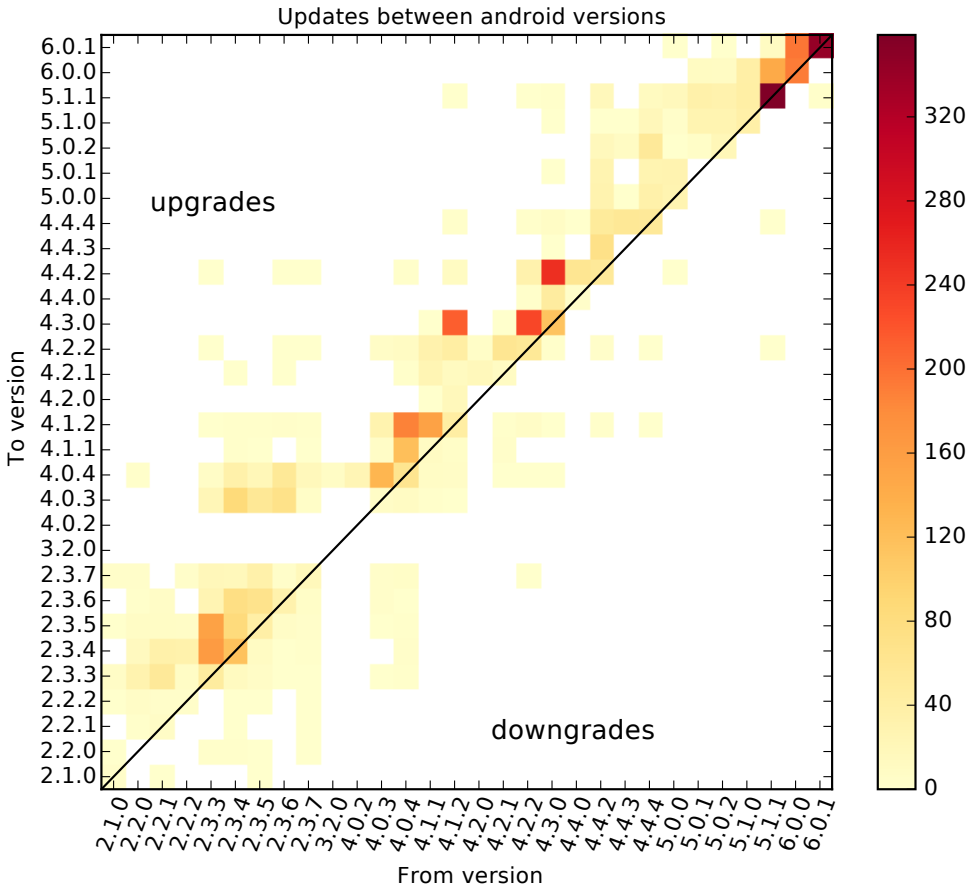


Figure 4.5: Updates between different Android versions in the Device Analyzer data

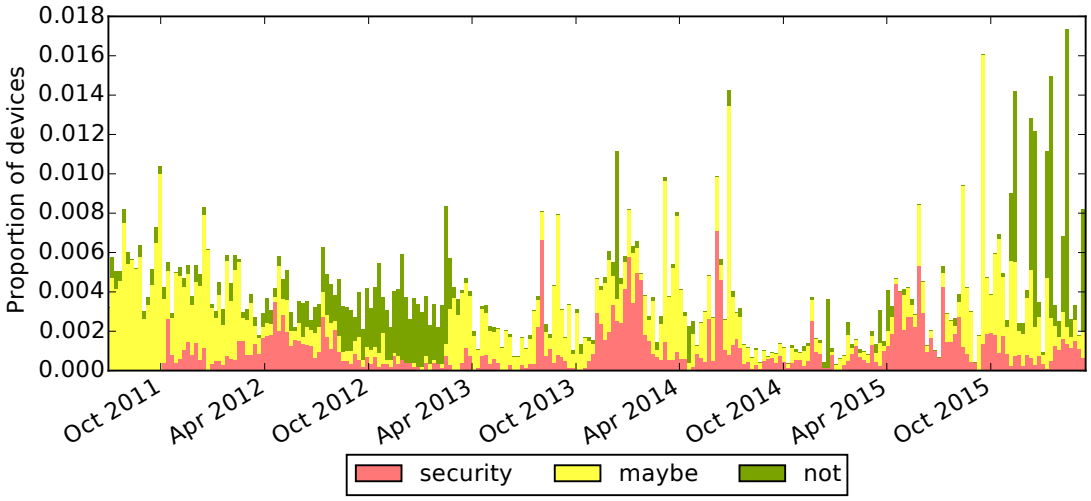


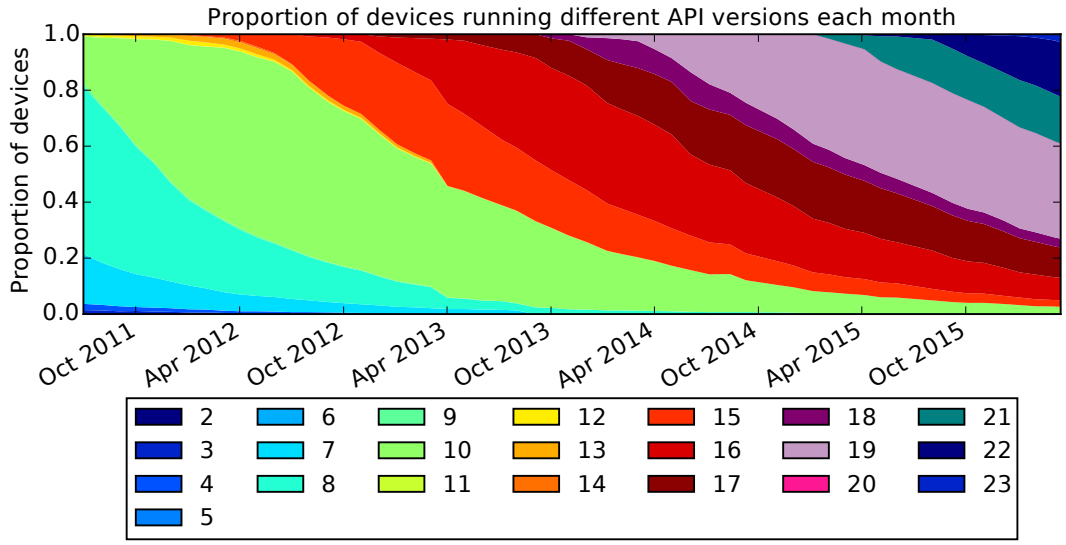
Figure 4.6: Proportion of updates each week that fixed or may have fixed security vulnerabilities

data Device Analyzer has collected, the number of updates received by a device per year is 1.43 ± 0.01. This compares badly with the number of critical vulnerabilities discovered per year of between 3.79 ± 0.84 (affecting all Android) and 7.96 ± 1.23 (including the device manufacturer specific ones).

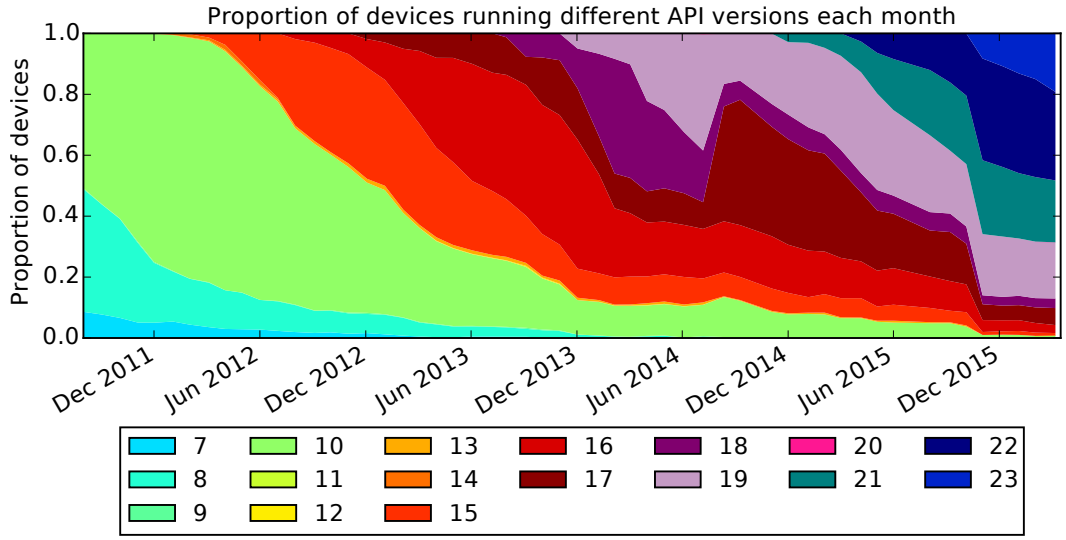
4.3 Threats to validity

4.3.1 Comparing Device Analyzer data with ground truth

The data from Device Analyzer that is used to investigate the proportion of devices exposed to different vulnerabilities is the OS version. Unfortunately there is no ground truth for OS version information. Google has published API version information almost every month since December 2009 and I have collated this information [231]. This information is essentially the ground truth for the Android API version distribution with data from all Android devices that use Google Play (about 1 billion in 2015). While API versions are too coarse grained to use for security update detection they are closely related to OS versions. If the Device Analyzer data on API versions are similar to the Google data on API versions then the Device Analyzer data on OS versions should be representative. Comparing the data from Google and from Device Analyzer shows them to be similar, except for the anomaly discussed later (§4.3.3). I analysed the difference between the API version data from Device Analyzer and Google Play, normalising for days since the API version was released (Figure 4.8). This shows that the Device Analyzer data systematically overestimates the prevalence of new API versions and underestimates the prevalence of old API versions (except for API version 17 that was particularly popular in a focused study discussed in the next section and so was temporarily overestimated when an old version). This means that the OS version information from Device Analyzer is likely to be overestimating the prevalence of new OS versions and hence the results presented in this chapter are a conservative estimate of the security of Android. Since most Device Analyzer users are self selecting and install Device Analyzer because they want to find out more about what their phone is doing or to aid research they may be biased and perhaps more likely to install updates.



(**a**) Google Play data on proportion of devices running different Android API versions.



(**b**) Device Analyzer data on proportion of devices running different Android API versions. The change in behaviour around August 2014 is explained in §4.3.3.

Figure 4.7: Monthly Android API version data

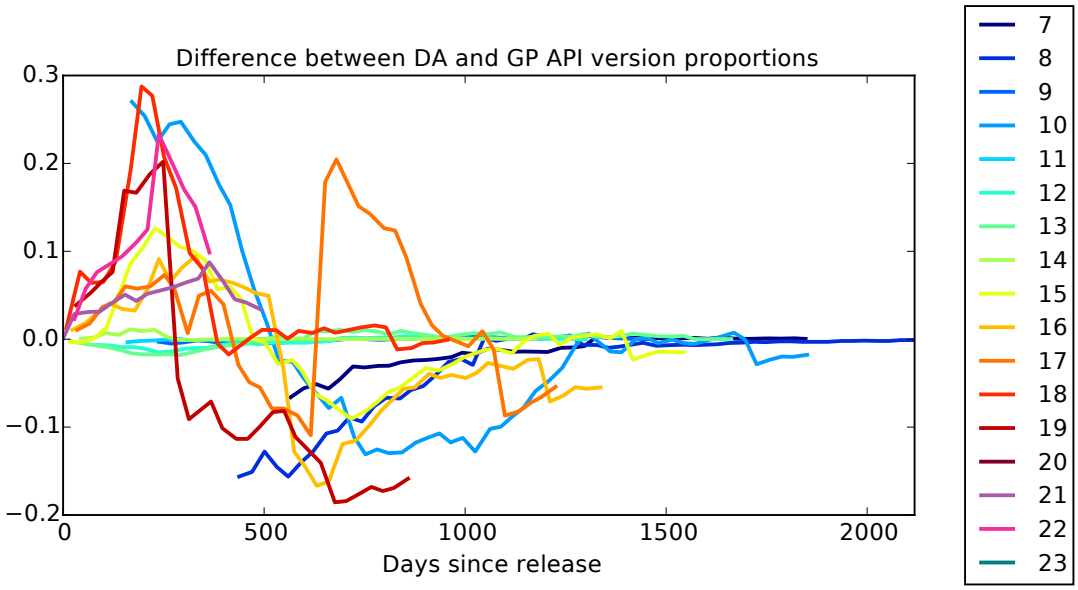


Figure 4.8: Difference between Device Analyzer and Google Play data on the proportion of devices running different Android API versions

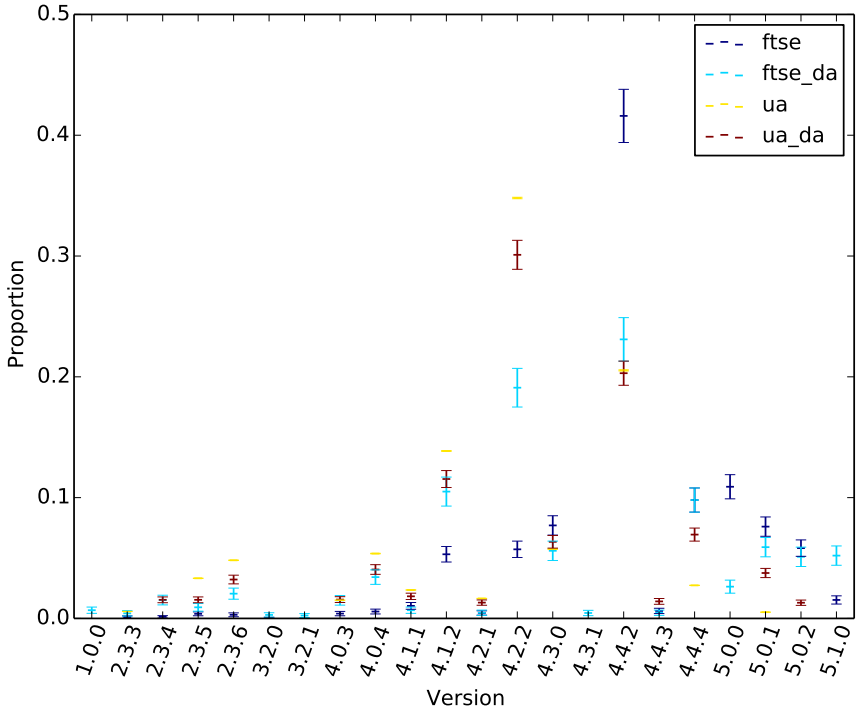


Figure 4.9: Comparison between FTSE, User-Agent and the corresponding Device Analyzer data, error bars indicate 95% confidence intervals.

4.3.2 Comparison with other data

Comparing the Device Analyzer data with two other sources of data shows they have similar distributions that bound the Device Analyzer data, indicating that it is likely to be typical. The comparable data sets are: data on 5 290 devices from a multinational FTSE 100 company's mobile device management database, which includes company and employee owned devices; and from 5 170 000 matching User-Agent headers on all HTTP traffic for 30% of Rwanda for a week.

The data from the FTSE 100 company for a week in April 2015 was used and the User-Agent data was collected in February 2015. Figure 4.9 shows the proportion of devices running each Android OS version in the two comparison data sets and the comparable periods from Device Analyzer. The general pattern this shows is that in the FTSE data newer versions are more popular than in the Device Analyzer data and that in the Rwanda data old versions are more popular. Therefore the Device Analyzer data on OS versions is bounded by these two data sets.

4.3.3 The effect of focussed studies on Device Analyzer

The Device Analyzer data is mostly generated by devices that have the Device Analyzer app installed because their owner happened to come across the app or as a result of news coverage of this research as there has been no advertising. However two collaborations with network operators lead to the network operators encouraging device owners to install Device Analyzer on their devices. This resulted in large numbers of new Device Analyzer users coming from particular network operators. One study was conducted in Norway with 654 installs and one in Bangladesh where 2 463 users installed Device Analyzer. This latter study is responsible for the sudden change in distribution of OS versions in Device Analyzer beginning in August 2014 and can be seen in Figures 4.1, 4.2 and 4.3. At the start of the study Bangladeshi users contributed over half of the Device Analyzer data. To investigate the sensitivity of the results to the Bangladesh study the data were truncated before this study started (all data after the start of the study ignored) and the percentage of vulnerable devices was found to be 85.6% rather than 87.6%, the percentage of devices running the latest version to be 5.71% rather than 5.67% and the number of updates per year to be 1.48 rather

than 1.43. Hence removing this anomaly has no substantial effect on the main conclusions.

4.3.4 Limitations

It is still not possible to know where new versions of Android come from. Over time newer versions of Android come to dominate, but the proportion due to new phone purchases verses the proportion due to updates of existing devices remains unknown. Device Analyzer is the only available large public data set with longitudinal traces of Android OS version strings. Unfortunately this dataset only records 4 080 upgrades that, broken down over 57 months and 30 OS versions is only 2.38 ± 0.03 updates per version per month. This is not enough to build a statistically significant prediction of the expected transition between versions of Android. This would require an average of at least 20 updates per version per month over a multi-year period from devices with multiple months of data. This is at least 10 times more data than is presently available from Device Analyzer.

If manufacturers shipped security updates without changing the build number then it would not be possible to detect this from the Device Analyzer data, but that would indicate a level of incompetence likely to damage security.

4.4 Related work

Using the methods and data described in this chapter I have determined that, on average, $87.6 \pm 0.0\%$ of devices were exposed to known critical vulnerabilities between 2011 and 2015. Felt et al. studied 6 Android handsets in 2011 and found they were exposed to root vulnerabilities at least 74% of the time [102]. My approach differs from their study because they used data from 6 handsets and assumed the best possible update distribution, while my work is based on a large sample of devices tracking the actual update distribution. Nevertheless, my own analysis as well as comparison with their work suggests protection against critical vulnerabilities has not improved substantially over the last 4 years. Felt et al. also found that 4 of the 46 malware samples (8%) they analysed contained root exploits, much lower than rates found in later, larger studies that found rates of 36.7% [271] and 40% [272] in 2012.

The percentage of Android devices running the most recent version ($5.67 \pm 0.0\%$) is much less than the rate ($> 90\%$) for Windows XP SP2 computers contacting the Microsoft update servers [116]. A simple numerical comparison is unfair because only one major OS version was considered in the Microsoft analysis, and data was only collected from computers contacting the update server, although this was the default. Later data demonstrates the difficulty of upgrading computers between major OS versions, with 27% of Windows computers running Windows XP in July 2014 [179], four months after Windows XP stopped receiving security updates and in June 2015 this figure was still 13% [180].

4.5 Summary

Modern mobile operating systems combine up to three layers of security protection to prevent app malware from taking control of a device. These are: the app store, checks performed during app installation, and the app sandbox. The app sandbox is the best place to prevent malware from gaining root. Unfortunately the Android sandbox is ineffective in the majority of cases: the latency in the security update process means that on average 87.6% of Android devices are exposed to known critical vulnerabilities that allow a malicious app to break out of the sandbox. Furthermore, only 5.67% of devices run the latest version of Android and devices apply 1.43 updates each year, less than the critical vulnerability discovery rate of between 3.79 and 7.96.

This chapter has shown how the Android OS version distribution changes and the frequency of updates. The open database of Android vulnerabilities (discussed in Chapter 3) has been used to determine the proportion of devices these vulnerabilities affected and how that changes over time. This has allowed the characterisation of the impact of critical vulnerabilities on the Android ecosystem as a whole. Through the quantification of the effectiveness of vulnerabilities and the length of time vulnerabilities remained available in the wild, and by examining the impact of vulnerabilities on individual devices over extended periods of time.

Despite the abundance of vulnerability in the Android sandbox, there has not been widespread compromise of Android devices and so the difficulty in getting malicious code into the sandbox in order to exploit it must be constraining

malware [124]. This indicates that the Google Play Store, the main entry point of apps may be effective at preventing malicious apps reaching devices, however as the next chapter shows, there are other ways that malicious code can reach devices.

CHAPTER 5

THE LIFETIME OF ANDROID API VULNERABILITIES: CASE STUDY ON THE JAVASCRIPT-TO-JAVA INTERFACE

The Android ecosystem today is a complex network of competing and collaborating companies. In this landscape, fixing security flaws is hard since it often involves many collaborating parties. This is particularly true for Application Programming Interface (API) vulnerabilities, where there is a flaw in the design of the interface used by third party apps. This chapter explores API vulnerabilities in Android and quantifies the rate at which these flaws are fixed on real devices.

Fixing API vulnerabilities is often hard: fixes may require changes to the API, which breaks backwards compatibility. The analysis described in §5.1 shows that an exponential decay function provides a good model for predicting the rate of fixes for API vulnerabilities in Android. Unfortunately, the rate of decay is low: it takes nearly a year for half of the Android devices using the Google Play Store to update to a new version of Android.

In order to ground this approach, a case study is included in §5.2 to investigate the timeline for fixing one API vulnerability in Android. The JavaScript-to-Java interface vulnerability was selected for this purpose as it is particularly serious and affects all versions of Android prior to the release of Android 4.2. This vulnerability allows untrustworthy JavaScript in a WebView to break out of the JavaScript sandbox, allowing remote code execution on Android phones; this can often then be further exploited to gain root access. While this vulnerabil-

ity is serious it does not meet the criteria for a critical vulnerability as discussed in Chapter 3 since it cannot be used directly to escalate privileges outside of the app sandbox. The fixing release was first available in October 2012 and as such there is now sufficient data to quantify the speed at which updates have propagated. While this vulnerability was first publicly disclosed in December 2012, the model predicts that the fix will not have been deployed to 95% of devices until August 2017, 4.82 years after the release of the fix. §5.2.2 shows that this vulnerability is exploitable between 0.6 ± 0.0 and 0.78 ± 0.10 times a day on Android devices.

The work presented in this chapter was published in the proceedings of Security Protocols XXIII [239] and resulted from collaboration with Thomas Coudray, Tom Sutcliffe and Adrian Taylor all working at Bromium. I combined data from the static analysis of APK files with data collected by Device Analyzer to produce the results presented here. I investigated several models for $f(t)$ and in discussion with Alastair R. Beresford decided on the simple one presented here as it is meaningful and an equally good fit to other more complex models. Thomas Coudray and Tom Sutcliffe collected the APK files and performed the static analysis.

5.1 API vulnerabilities in Android

At the beginning of 2015, the Android had been revised twenty one times since the first version was released in 2008. I have manually collected the monthly statistics[1] of the proportion of devices using each API version when the devices connect to the Google Play Store that Google has published since December 2009 [231]. These statistics are plotted in Figure 5.1. The API version distribution shows a clear trend in which older API versions are slowly replaced by newer ones. API version data from Google Play rather than Device Analyzer is used because the Google Play data represents the ground truth for API version data and for this analysis unlike the other analyses the greater detail provided by Device Analyzer is not required.

In order to quantify the lifecycle of a particular API version the data is re-

[1]They only list the statistics for the current month on their website and so I had to find news reports about their results for each month or use archive.org to find the data, now I visit the page each month [138].

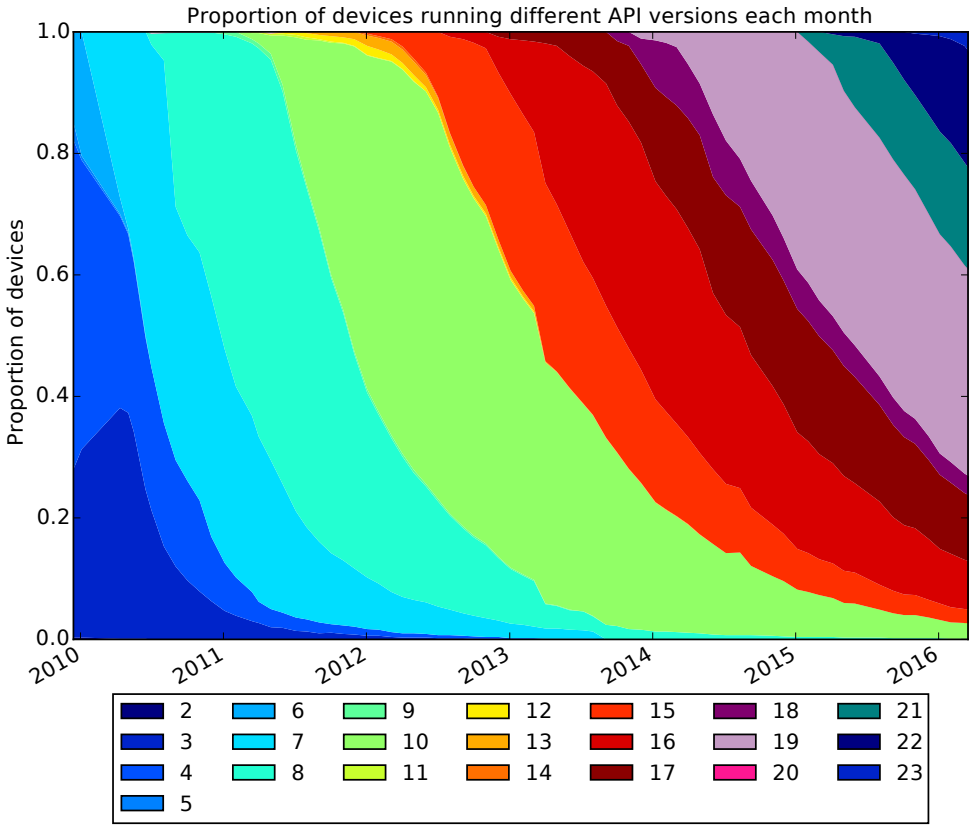


Figure 5.1: Proportion of devices running different API versions

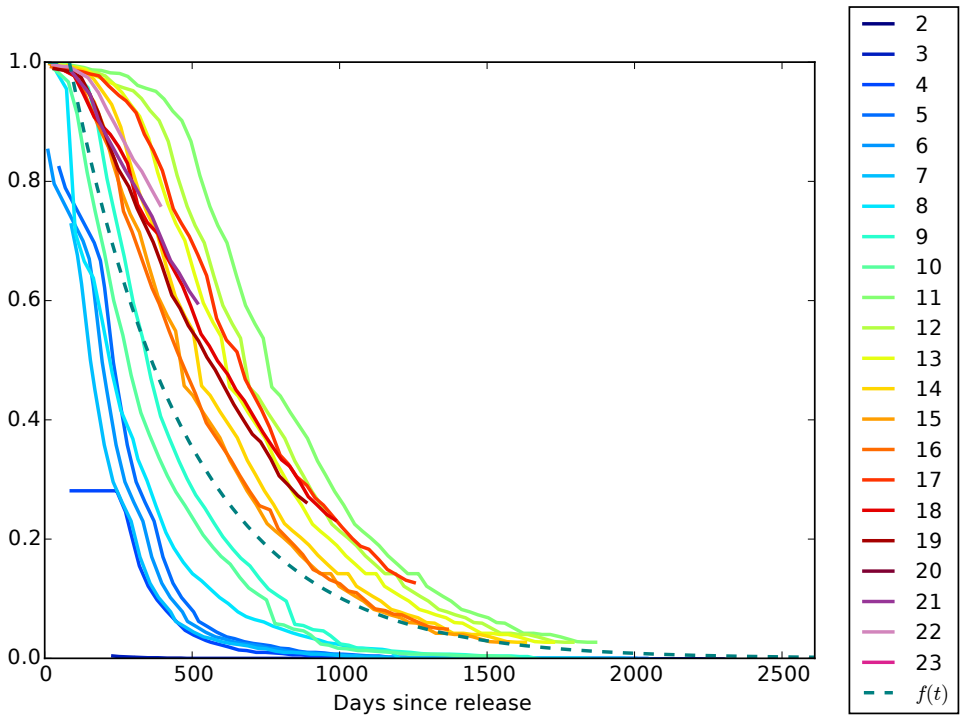


Figure 5.2: Proportion of devices not updated to particular versions of Android or any later version. The best fit $f(t)$ is an exponential decay function.

processed in two ways. Firstly, in order to understand the speed of adoption of a particular version of the API, it is normalised for the number of days since release of the API version ($x = \text{day} - \text{releaseDate}$). Secondly, rather than plotting the proportion running a particular version, the proportion of devices that have *not* upgraded to a particular API version or any successor is plotted. This gives the proportion not supporting a particular API version. For example, when a new API version is first released, no devices could have already been updated to it and therefore the proportion that have not upgraded is 1. As devices begin to upgrade to the new API version (or any subsequent version), the proportion not upgraded tends to 0.

Data from Figure 5.1 is replotted in Figure 5.2 to show the proportion of devices not upgraded to a particular version of Android against days since the API version was first released. These data show that all API version upgrades follow a similar trend: a slow initial number of upgrades in the first 250 days, then widespread adoption between 250 and 1000 days, followed by a slow adoption of the new API version by the remaining devices.

Visually, these data appear to have an exponential decay as it tends to zero. Fitting a model to these data allows predictions about future behaviour to be made. Therefore decay is modelled as $f(t)$, a combination of an exponential function together with a delay t_0 that offsets the start time:

$$f(t) = \begin{cases} 1.0 & \text{if } t < t_0 \\ e^{-\text{decay}(t-t_0)} & \text{otherwise} \end{cases} \tag{5.1}$$

Fitting $f(t)$ to these, gives a Root-Mean-Squared-Error (RMSE) of 0.167 with the parameters $t_0 = 83.6$ days, $\text{decay} = 0.002\,49$ days^{-1} across all API versions. An RMSE of 0.167 compares favourably with a standard polynomial fit (a 3 degree polynomial fit gave an RMSE of 0.167) or a spline fit (RMSE of 0.167) and gives a meaningful model of behaviour rather than a generic curve.

From this fit, the number of days from the release of a new version of Android until 50% of devices are running that version or higher is 362 (0.991 years) and full deployment to 95% of devices takes 1 290 days (3.52 years). The same analysis using the Device Analyzer data on OS versions in use gives 332 days (0.908 years) and 1 160 days (3.19 years) respectively, which is faster, but not by much.

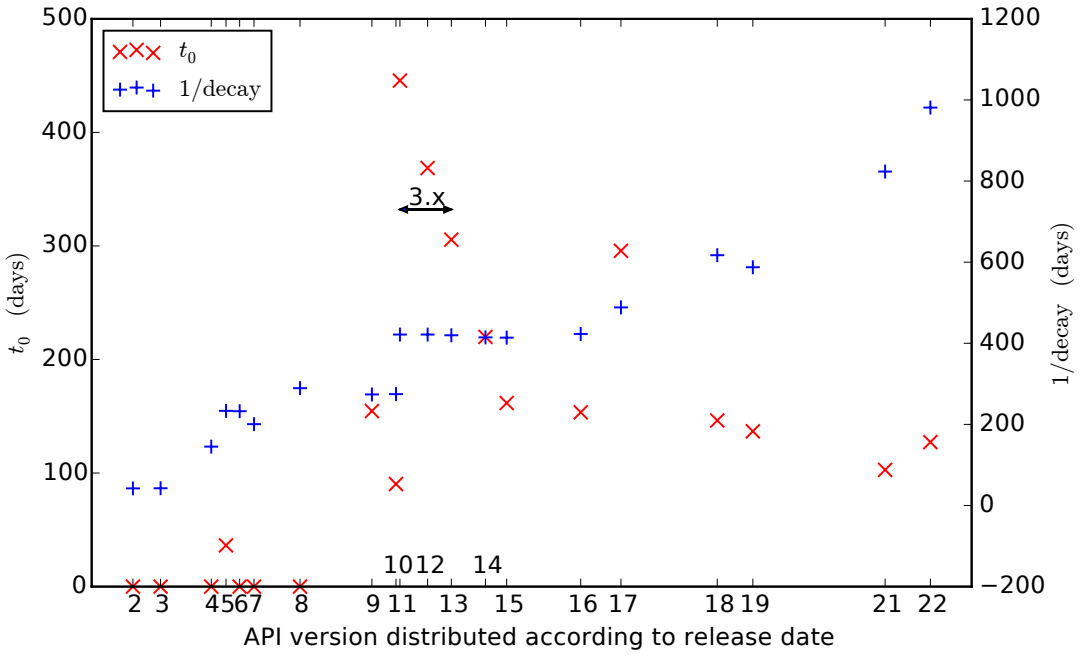


Figure 5.3: Fitted parameters for different API versions.

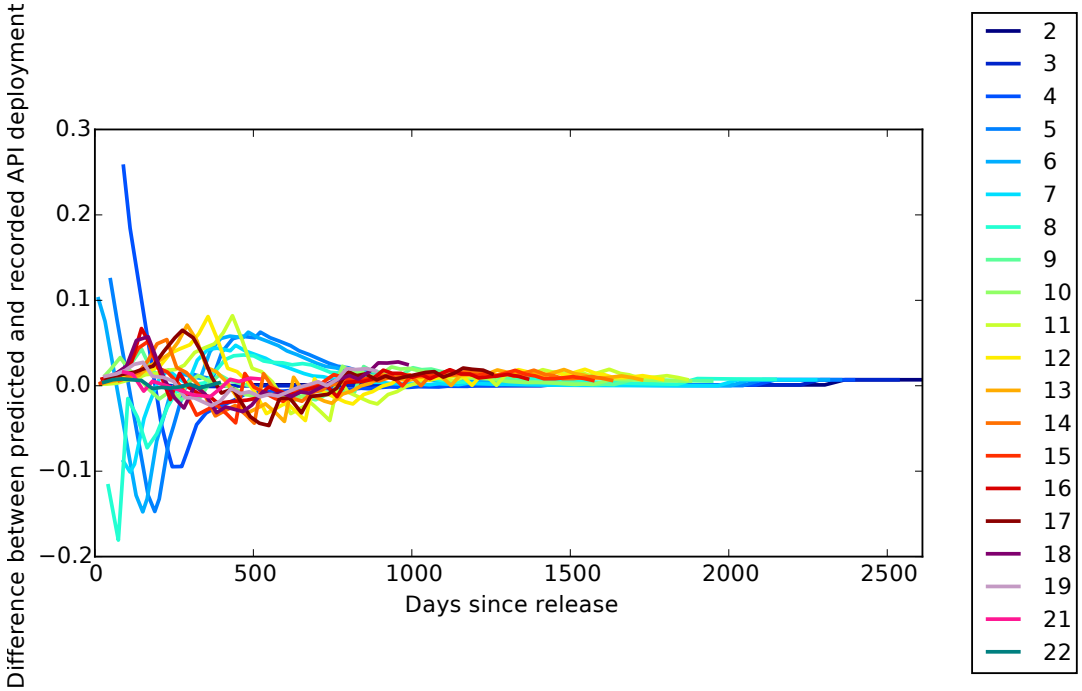


Figure 5.4: Difference between predicted behaviour and recorded behaviour.

Unfortunately, while this is a good predictor of average behaviour, individual API versions are systematically different from each other. However, the fit parameters from the global analysis can be used to seed a fit for each API version. This produces the parameters in Figure 5.3 with t_0 and 1/decay plotted as this means that larger values for both are worse. API versions 11, 12 and 13 represent Android 3.x, which never saw widespread deployment because these versions targeted tablets and were not available for use on phones. Discounting those values, Figure 5.3 shows a trend of updates taking longer over time as t_0 increases and 1/decay increases. This implies that the Android ecosystem is getting worse at over time distributing updates.

The differences between predictions and recorded reality is shown in Figure 5.4. It shows the difference between the prediction and recorded behaviour oscillating around 0 with some systematic errors early on due to the simple model of $f(t)$. The errors are mostly less than 10% and fall over time.

5.2 Case study: The JavaScript-to-Java interface vulnerability

The Android *WebView* provides a developer with a web browser UI component that can be controlled programmatically by a hosting app, including rendering dynamic HTML content driven by JavaScript. To allow convenient interaction between the WebView and the hosting app, a Java object instance can be bound to a JavaScript variable name, allowing the JavaScript code to call any public methods on the Java object. Prior to Android 4.2, the exposed public methods included those inherited from parent classes, including the `getClass()` method of `java.lang.Object`. This permitted the execution of arbitrary Java code from the JavaScript running inside the WebView. For example, Java reflection controlled from JavaScript can be used to execute Linux programs such as `id`, as shown in Figure 5.5. This is a security vulnerability (CVE-2012-6636) that can be used to remotely run malicious code in the context of an app using the JavaScript-to-Java interface vulnerability and from there exploit other vulnerabilities to gain root privileges on devices and spread as an Android worm.

The attack is comprised of the following steps.

1. Content for WebViews in apps commonly comes from untrustworthy sources

```
<script>
    Android.getClass()
        .forName('java.lang.Runtime')
        .getMethod('getRuntime',null)
        .invoke(null,null).exec(['id']);
</script>
```

Figure 5.5: JavaScript attack, assuming *Android* is the JavaScript alias for the exposed Java object.

	target API < 17	target API ≥ 17
API < 17	Vulnerable	Vulnerable
API ≥ 17 and OS $< 4.4.3$	Vulnerable	Safe
OS $> 4.4.3$	Safe(ish)	Safe

Table 5.1: The different categories that uses of `addJavascriptInterface` can fall into depending on the target API of app and the Android version of the device.

or over an unauthenticated HTTP connection, so an active attacker controlling the network (strategies for doing this are discussed in Section 5.2.1) can inject a malicious JavaScript payload into the HTTP stream, which is then executed inside the JavaScript sandbox.

2. The malicious JavaScript can then use the JavaScript-to-Java interface vulnerability to break out of the JavaScript sandbox into the app context.

3. Then the malicious code can often use other known vulnerabilities to break out of the app sandbox and gain root privileges on the device. On average, approximately 88% of Android devices are vulnerable to at least one known root vulnerability (Chapter 4).

4. Once an attacker has root on a device, he can use ARP spoofing or ICMP redirect attacks to reroute local traffic through the device and inject malicious JavaScript into any HTTP traffic, thereby starting step (1) above on a new device. Thus the attack has the potential to act as an Android worm.

Google has modified the Android API or its implementation twice in an attempt to fix the JavaScript-to-Java interface vulnerability. In the first fix, the function of the JavaScript-to-Java interface was modified in Android 4.2 to ensure that only public methods with the annotation `@JavaScriptInterface` could

be called from within JavaScript for new apps. This change only prevents the attack if *both* the phone is running Android 4.2 or greater *and* the app has been compiled with a recent version of the Android framework with a target API Level of 17 or newer. In the second fix, for devices using the WebView based on Google Chrome for Android version 33.0.0.0 or later (included in Android 4.4.3 and later), access to the `getClass` method on injected Java objects was blocked.[2] This prevents the most obvious JavaScript-to-Java interface attacks by preventing direct access to the Java runtime. An attacker must instead find a different route through the public methods on the injected Java objects, which may not always exist and is certainly much harder. This situation is shown in Table 5.1.

Any app with a WebView containing a JavaScript-to-Java interface is potentially vulnerable to this attack. An app that uses the JavaScript-to-Java interface is labelled *always vulnerable* if it contains a target API level of 16 or older, since such an app is vulnerable when run on any version of Android less than 4.4.3; and *vulnerable only on outdated devices* if the app has a target API Level of 17 or newer, since such an app is vulnerable only if running on a device running Android 4.1.x or older.

5.2.1 Injection threat model

There are several different scenarios in which an attacker could inject malicious JavaScript to exploit the JavaScript-to-Java interface vulnerability.

1. An attacker could control the original server that supplied 'legitimate' HTML either through compromising it or by using some other means (such as buying ads) to supply the malicious JavaScript.

2. An attacker could control a node on the path from the original server allowing them to inject malicious JavaScript into the HTTP traffic. This would involve gaining control of a node or connecting a new node between two existing nodes.

[2] `https://codereview.chromium.org/213693005/patch/20001/30001` committed on 2014-04-04 by mnaganov as 261801 or afae5d83d66c1d041a1fa433fbb087c5cc604b67 or e55966f4c3773a24fe46f9bab60ab3a3fc19abaf "[Android] Block access to java.lang.Object.getClass in injected Java objects" fixing bug 359528.

3. An attacker could control traffic passing through the device's local network and inject malicious JavaScript. This could be achieved by either running a public Wi-Fi network, or compromising an existing network using ARP spoofing or ICMP redirect attacks to redirect all traffic via a machine under their control. This can be done opportunistically and does not require physically connecting a device or compromising and existing device.

Level 1 attacks can be mitigated by better system security and input validation at the original server. Level 2 and Level 3 attacks can be mitigated by apps using HTTPS with proper validation of certificates [99] (for example using pinning [52]) to prevent an attacker from injecting malicious JavaScript. Level 3 attacks can also be mitigated through the use of a secure VPN to a trustworthy network and by better security on the local network, such as, protection against ARP spoofing, ICMP redirect attacks and independently encrypted connections to the router.

5.2.2 Sources of vulnerability

To investigate the severity of this vulnerability, data on which apps use the JavaScript-to-Java interface and where the apps use it is required. Bromium analysed 102 174 APK files from the Google Play Store collected on 2014-03-10 and between 2014-05-10 and 2014-05-15. They found that 21.8% (22 295) of apps were always vulnerable, 15.3% (15 666) were vulnerable only on outdated devices, 62.2% (63 533) were not vulnerable and 0.67% (680) could not be analysed due to failures of their static analyser. These results are presented in Table 5.2 and show that most apps are not vulnerable, but that more apps are always vulnerable than are vulnerable only on outdated devices.

The static analysis was performed by decompiling the APKs using `apktool` and extracting the target API version from the Android Manifest. Apps using JavaScript-to-Java interface were detected by string matching for '`addJavascriptInterface`' in the decompiled .smali files.

Of the 38 000 vulnerable apps, 12 600 were found in the Device Analyzer data [246]. Those that are not in the Device Analyzer data are unlikely to be widely used, since these apps were not installed on any of the 24 600 devices in Device Analyzer data.

Classification	Percentage	Count
Always vulnerable	21.8	22 295
Vulnerable only on outdated devices	15.3	15 666
Not vulnerable	62.2	63 533
Unscannable	0.67	680

Table 5.2: Percentage of the 102 174 apps analysed that fell in each category

Based on the Device Analyzer data, always vulnerable apps were started 0.6 ± 0.0 times a day between the disclosure of the vulnerability and the start of Bromium's APK file collection, with 8.34 ± 0.67 such apps installed.

Apps vulnerable only on outdated devices were started 0.78 ± 0.10 times a day between the disclosure of the vulnerability and the start of Bromium's APK file collection, with 7.27 ± 0.71 such apps installed.

Hence, on an outdated device, vulnerable apps were started 1.38 ± 0.11 times a day with 15.6 ± 0.9 vulnerable apps installed. Due to the fact that not all the apps are observed by Device Analyzer, these rates are likely to be underestimates. It is also possible that the Device Analyzer data could be biased towards users with more apps than is typical, which might cause this figure to be an overestimate.

5.2.3 Lifetime of the vulnerability

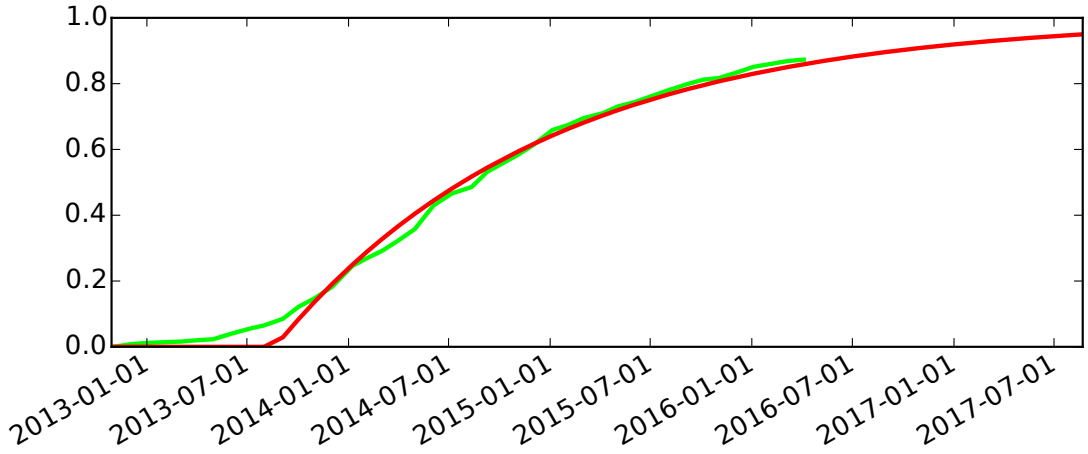


Figure 5.6: Proportion of fixed devices with these data from Google Play given in green and above it the prediction in red

The vulnerability was first publicly recorded in December 2012 [25]. The proportion of devices that contacted the Google Play Store and are secure for

apps that use the new API are shown in green in Figure 5.6. In summary, in April 2016 87.3% of devices were running a version of Android that protects users from apps vulnerable only on outdated devices.

This vulnerability will cease to be problematic when all Android devices run API version 17 or later *and* all apps that use JavaScript-to-Java interface target API version 17 or later. The model for $f(t)$ from Equation 5.1 and knowledge that API version 17 was released in October 2012 predicts 95% of all Android devices to be secure for apps vulnerable only on outdated devices by August 2017. This prediction is shown in red in Figure 5.6. Without visibility into the way apps' target API versions on Android change over time it is harder to understand whether always vulnerable apps will continue to represent a significant risk after almost all Android devices support API version 17 or later.

5.2.4 Solutions

There are various strategies that could have been adopted to more rapidly mitigate this vulnerability. Android could have broken API compatibility and backported the fix to all versions of Android, however as shown in Chapter 4, security updates are deployed slowly on Android. Android could refuse to load JavaScript over HTTP and require HTTPS (with verified certificates) or use local storage, which would make MITM attacks injecting malicious JavaScript harder. Part of the problem is that the libraries (particularly ad-libraries) that developers bundle inside their apps target old API versions and developers need to update their dependencies to versions that target higher versions.

If Android had a more comprehensive package management system, which handled dependencies, then apps could be loosely coupled with their ad-libraries and the libraries could be updated to fixed versions without the app developers having to re-release every app that used it. Alternatively, to maintain backwards compatibility while fixing the vulnerability, apps could be automatically rewritten to fix the vulnerability [265].

Users could use a VPN to tunnel all their traffic back to a trustworthy network so that MITMs on local networks (such as open Wi-Fi access points) would not be able to mount this attack, but this would not protect against attackers on the network path to the ad-server, or malicious ad-servers.

The fix included in Android 4.4.3 discussed at the beginning of §5.2, where

access to the `getClass` method is blocked, substantially mitigates this vulnerability, but for some apps there may be other exploit paths. Language solutions such as Joe-E could be used to enforce security properties, including preventing JavaScript from executing arbitrary code [169]. Such a solution would need to avoid legacy interfaces (that the JavaScript-to-Java interface vulnerability illustrates the danger of) allowing this protection to be bypassed [248].

5.3 Related work

The JavaScript-to-Java interface vulnerability has been investigated before. It was demonstrated by MWR Labs [149] who showed how it could be used to run the Android security assessment framework `drozer`, which can be used to run a remote shell on a compromised Android device. The strategies Bromium used for statically analysing Dalvik bytecode to discover use of JavaScript-to-Java interface have also been used previously [258].

Attacks have been published against WebView [160] including those relating to the JavaScript-to-Java interface and vulnerabilities caused by the violation of the origin-based access control policy in hybrid apps [115].

There have been investigations of the behaviour of ad-libraries on Android. Stevens et al. demonstrated how attacks could be mounted on JavaScript-to-Java interface used by ad-libraries, but without realising the significance of `getClass` [224]. However, unlike the JavaScript-to-Java interface vulnerability, these attacks continue to work on fixed devices even for apps vulnerable only on outdated devices. Grace et al. have shown that ad-libraries require excessive permissions and expose users to additional risks [127], which are further compounded by the JavaScript-to-Java interface vulnerability.

To counteract the problems caused by ad-libraries being packaged within an app, and thereby inheriting all their permissions, there have been proposals to separate out the ad-libraries into separate processes by altering Android to provide an API [187] and then automatically rewriting apps to use such an API [211]. This improves security, particularly if it means that the ad-libraries can be updated independently of the apps, but it does not otherwise help if an attack on JavaScript-to-Java interface can be followed up with a root exploit.

The PlayDrone crawler was able to analyse over 1 100 000 apps, many more

than were used in this analysis [244] but this work could be repeated with those data.

5.4 Summary

This chapter proposed the exponential decay model for Android API vulnerabilities and explored one case study: the JavaScript-to-Java interface vulnerability. Applying this model to this case study showed that for apps that are vulnerable only on outdated devices, 95% of all Android devices will be protected from the JavaScript-to-Java interface vulnerability by August 2017, 4.82 ± 0.97 years after the release of the fix. It is not known whether always vulnerable apps will continue to present a security risk and therefore it is unclear whether Android users will be safe from this vulnerability after this date. This chapter presented a detailed discussion of the JavaScript-to-Java interface vulnerability including how it could be used to produce an Android worm and Google's multiple attempts to fix it. The injection threat model was presented along with a discussion of the possible solutions. Analysis of apps allowed the exploitability of the JavaScript-to-Java interface vulnerability to be demonstrated.

The previous chapter argued that the Play Store and Verify Apps must be what is keeping Android secure as the app sandbox is so often exposed to vulnerabilities, but this chapter has demonstrated that other vulnerabilities which allow remote code execution are also slow to fix and bypass the those protections. The next chapter will present a different approach to fixing the problem.

CHAPTER 6

SECURITY METRICS FOR THE ANDROID ECOSYSTEM

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind.
>
> Lord Kelvin, 1883

The security of Android depends on the timely delivery of updates to fix critical vulnerabilities, which Chapter 4 showed were lacking and Chapter 5 showed were necessary. This chapter maps out the complex network of players in the Android ecosystem who must collaborate to provide updates, and determines that inaction by some manufacturers and network operators means many handsets were vulnerable to critical vulnerabilities. It defines the FUM security metric to rank the performance of device manufacturers and network operators, based on their provision of updates and exposure to critical vulnerabilities. Where a FUM score of 0 indicates that the device is always vulnerable to many vulnerabilities and never runs the latest version while 10 indicates that the device is never vulnerable to any vulnerabilities and always runs the latest version. A corpus of 24 600 devices from Device Analyzer is used to show that there is significant variability in the timely delivery of security updates across different device manufacturers and network operators. This provides a comparison point for purchasers and regulators to determine which device manufacturers and network operators provide security updates and which do not. Android as a whole is assigned a FUM security score of 2.71 out of 10. In these data, Nexus devices (produced by Google in collaboration with device manufacturers) do consider-

ably better than average with a score of 5.63; and LG is the best manufacturer with a score of 4.28 (it produced several Nexus devices).

All large software systems today contain undiscovered security vulnerabilities. Once discovered, these flaws are often exploited, and therefore the timely delivery of security updates is important to protect such systems, particularly when devices are connected to the Internet and therefore can be exploited remotely. Manufacturers and software companies have known about this issue for many years and are expected to provide regular updates to protect their users. For example, Windows XP could be purchased for a one-off payment in October 2001 and received monthly security updates until support ended in April 2014.

Unfortunately something has gone wrong with the provision of security updates in the Android market. Many smartphones are sold on 12–24 month contracts, and yet the Device Analyzer data show few Android devices receive many security updates, with an overall average of just 1.43 updates per year, leaving devices unpatched for long periods of time.

To understanding why, more information about the Android ecosystem as a whole is required. It is a complex system with many parties involved in a long multi-stage pipeline [137]. §6.1 maps out and quantifies the major parties who must collaborate to provide updates and §6.4 shows that inaction by some manufacturers and network operators means many handsets were vulnerable to critical vulnerabilities. Understanding this ecosystem is all the more important because device manufacturers have introduced additional vulnerabilities in the past [126].

Corporate and public sector buyers are encouraged to purchase secure devices, but there is little concrete guidance on the specific makes and models providing timely security updates. For example, CESG, which advises the UK government on how to secure its computer systems, recommends picking Android device models from device manufacturers that are good at promptly shipping security updates, but it does not state which device manufacturers these are [46] and at time of writing CESG had only certified one Android device model [47]. Similarly, I am collaborating with a FTSE 100 company who wish to know which devices are non-vulnerable and which manufacturers provide updates.

The difficulty is that the market for Android security today is like the market for lemons: there is information asymmetry between the manufacturer, who knows whether the device is currently non-vulnerable and will receive security

updates, and the customer, who does not. To address the asymmetry, this chapter proposes a scoring system and provide numbers on the historic performance of device models found in the Device Analyzer [246] project. It proposes three metrics: f the proportion of running devices free from critical vulnerabilities over time; u the proportion of devices that run the latest version of Android shipped to any device produced by that device manufacturer; and m the mean number of outstanding vulnerabilities affecting devices not fixed on any device shipped by the device manufacturer. From this the composite FUM score is derived, which is hard to game (§6.8).

The FUM score enables corporate and public sector buyers, as well as individuals, to make more informed purchasing decisions by reducing the information asymmetry. The FUM score also supports better regulation, and indeed at time of writing there there was ongoing legal action to force network operators to ship updates for security vulnerabilities [215].

In August 2015, in the wake of substantial press coverage of the Android Stagefright remote code execution as root vulnerability – both Google [159] and Samsung [206] promised to release security updates monthly. While the industry was aware of this research at that point this is not a direct impact of this research. However, this work will make it possible to measure the impact of this change and will incentivise manufacturers that have not yet made this commitment.

The work presented in this chapter was published at the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) [236]. Andrew Rice suggested computing a score, computing it as a function of time and testing the sensitivity of the scoring metric; many of the ideas in this work resulted from discussion with Alastair R. Beresford; and an anonymous SPSM reviewer suggested considering utilitarianism and using Spearman's rank rather than my earlier idea of using the Damerau-Levenshtein string-edit distance metric [19]. I investigated various possible scoring metrics and selected the one used, the analysis and presentation are all my own work. A summary of work in this chapter combined with information from the introduction and background chapters has been accepted for publication at the Internet of Things Software Updates workshop 2016 [234].

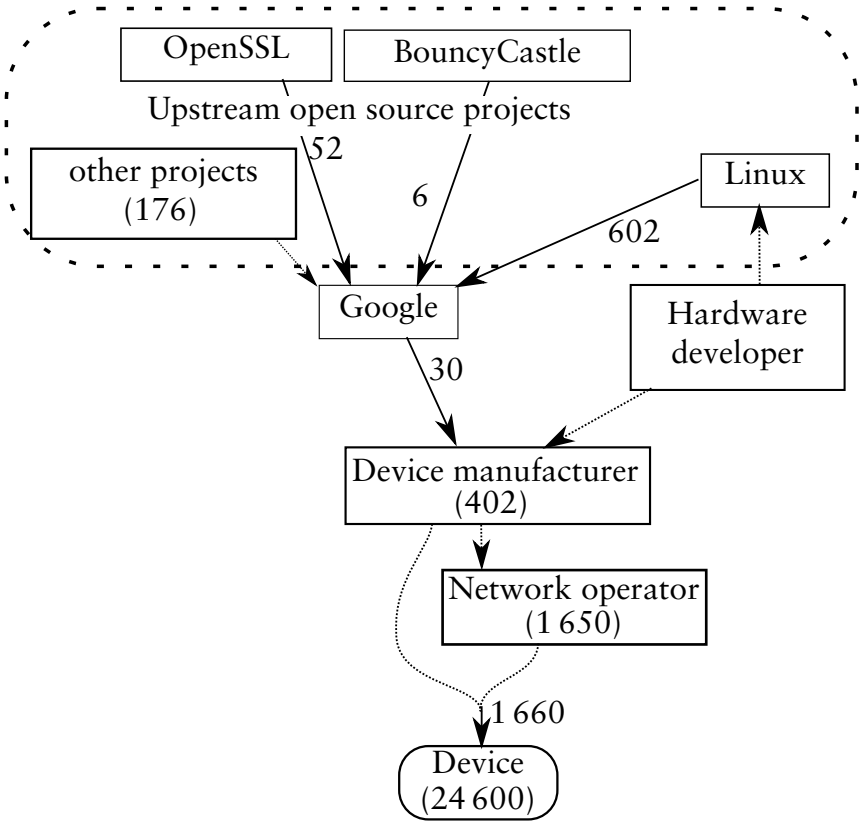


Figure 6.1: Flow of updates between participants in the Android ecosystem. Numbers on edges indicate updates shipped between July 2011 and March 2016, those in brackets represent number of such entities. Dotted arrows indicate flows that cannot be measured because no public data is readily available.

6.1 Android Ecosystem

This section describes how the Android ecosystem functions and how Android versions are produced by using Device Analyzer data and analysing the Android source code and upstream projects. It quantifies the number of updates shipped by various entities in the ecosystem and the number of entities.

To understand how vulnerabilities in Android are fixed consider the Android update process, modelled in Figure 6.1. There are five entities or groups that contribute towards Android updates: the network operators, the device manufacturers, the hardware developers, Google, and the upstream open-source projects. Android builds on various open-source projects, such as the Linux kernel, OpenSSL and BouncyCastle cryptography libraries. Consequently Android can include any compatible versions of those projects, including those that fix

security vulnerabilities. Android also incorporates various drivers for different bits of hardware. The Android platform is then built from these components by Google. The code for each Android release or update is kept secret until after a binary release has been published [121]. Device manufacturers receive advanced access in order to prepare handsets. The network operator may then make or request customisations and perform testing before shipping the update to the device. Sometimes device manufacturers ship updates directly to the user without involving the network operator. Sometimes the device manufacturer and Google collaborate closely to make a particular phone, such as with Nexus devices, enabling Google to ship directly to the device. Sometimes device manufacturers incorporate upstream open-source project releases directly, and sometimes incorrectly – for example previous work has recorded evidence of broken nightly builds of sqlite in Android releases on some device models [246].

The numbers of devices (24 600), network operators (1 650) and device manufacturers (402) in Figure 6.1 come from the Device Analyzer data. Device manufacturer and network operator counts were obtained by normalising the results reported by Android to Device Analyzer of the device manufacturer and active network operator. This normalisation is a manual task that involves removing invalid values (e.g. 'manufacturer' or 'airplane mode is on'), collating across company name changes (e.g. 'lge' to 'LG'), normalising punctuation, removing extra strings sometimes added (e.g. '(2g)' or 'communications') and mapping some incorrectly placed model names back to their manufacturer. This normalisation is not perfect and so these are overestimates on the Device Analyzer data but these estimates are likely still underestimates as there will be some device manufacturers and network operators that are not included in the Device Analyzer data. The representativeness of Device Analyzer is discussed in §4.3.1.

In Figure 6.1 the number of updates received by devices (1 660) is the number of different full version strings observed in Device Analyzer. The number of updates shipped by Google (30) is the number of Android versions reported in Device Analyzer that affected more than 1% of devices for more than 10 days. This significance test is to remove spurious versions recorded in Device Analyzer such as '5.2.0' in 2012 that had still not been released at time of writing.

To investigate the influence of external projects on Android, data about them was collected, these data and the scripts that generated them is available from AndroidVulnerabilities.org (AVO) [232]. These scripts analysed the Android

Project	# releases	latency (days)
Linux	602	137 ± 48
OpenSSL	52	108 ± 63
Bouncy Castle	6	220 ± 70

Table 6.1: Flow of updates from upstream projects into Android. Number of updates as in Figure 6.1, latency in days between the upstream release and the release of the first Android version containing it, for all pairs of versions I have data on.

Open Source Project's source tree to examine the source code of each of the external projects to find the project version associated with each Android version tag on the repository. There were 176 external open-source projects in Android, contributing 25 million lines of code. The top 40 by lines of code (99.7% of the total) were analysed and I was able to automatically extract the versions of those projects included in different versions of Android for 28 of these (24.9% of the total). This found 72 distinct versions, a median of 2.0 and mean of 2.57 ± 1.84 versions per project. Android rarely changes the version of external projects it includes.

To compute the latency between upstream releases and the release of the first version of Android containing that release, I scraped the release pages to obtain the version numbers and release dates. This allows the computation of the latency between an upstream project release and inclusion in Android, this is shown in Table 6.1. The versions included in Android were about half a year old when the first version of Android containing it was released.

6.2 Method: Scoring for security

Computing how good a particular device manufacturer or device model is from a security standpoint is difficult because it depends on a number of factors that are hard to observe, particularly on a large scale. Ideally, both the prevalence of potential problems that were not exploited and actual security failures would be considered. However, in the absence of such data, this section proposes a scheme for assigning a device a score out of ten based on data that can be observed, is based on previous metrics, and that should correlate with the actual security of the devices.

The FUM score is computed from three components:

free f The proportion of running devices free from critical vulnerabilities over time. This is equivalent to $1-$ Acer and Jackson's proposal to measure the security based on the proportion of users with at least one unpatched critical vulnerability [2] and similar to the Vulnerability Free Days (VFD) score [260]. Unlike VFD, this is the proportion of running devices that were free from critical vulnerabilities over time, rather than the number of days that the device manufacturer was free from outstanding critical vulnerabilities, as that does not take account of the update process.

update u The proportion of devices that run the latest version of Android shipped to any device produced by that device manufacturer. This is a measure of internal updatedness, so a low score would mean many devices are left behind. This assumes that newer versions are better with stronger security. Historically, steps have been taken to improve Android security in newer versions so this assumption should generally hold, but sometimes new updates introduce new vulnerabilities.

mean m The mean number of outstanding vulnerabilities affecting devices not fixed on any device shipped by the device manufacturer. This is related to the Median Active Vulnerabilities (MAV) measure [260] but is the mean rather than the median, since this gives a continuous value. An example is given in Figure 6.2.

These three metrics f, u and m, together measure the security of a platform with respect to known vulnerabilities and updates. The value f is a key measure of the direct risk to users as a known, unfixed, vulnerability means devices are vulnerable. However, it does not capture the increased risk caused by multiple known vulnerabilities, which gives an attacker more opportunities and increases the likelihood of a piece of malware having a matching exploit. This is captured by the m score, which measures the size of the device manufacturers queue of outstanding vulnerabilities. The m score does not take into account the update process or measure actual end user security. Neither of these metrics capture whether devices are left behind and not kept up-to-date with the most recent (and hopefully most secure) version, which is captured by u.

A score out of 10 is provided as this is easy for phone buyers to understand, because many ratings are given as a score out of 10. Since f is the most important

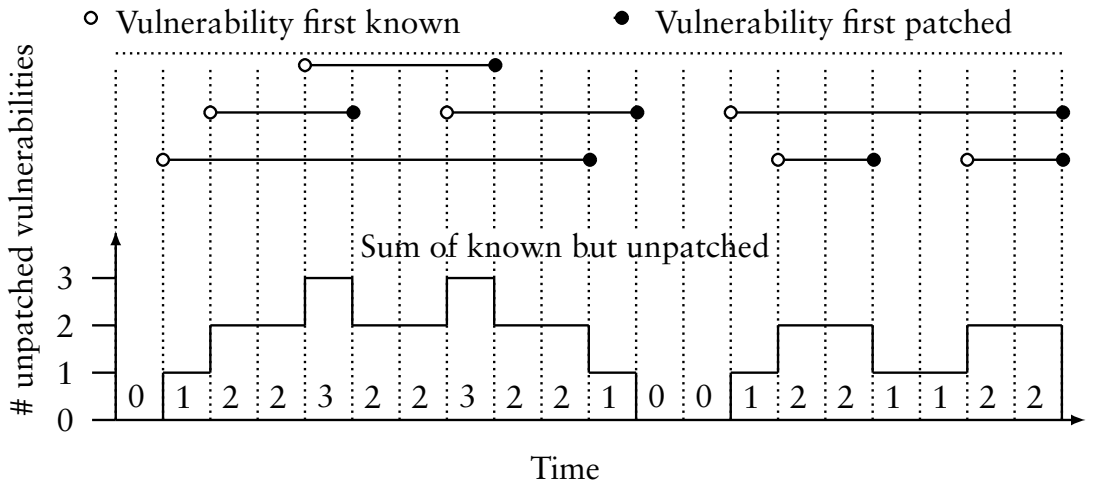


Figure 6.2: As vulnerabilities are discovered and patched the sum of known but unpatched vulnerabilities each day varies. From this m can be calculated: $m = (0 \times 3 + 1 \times 5 + 2 \times 10 + 3 \times 2)/20 = 1.55$. For comparison VFD = 0.15 and MAV = 2. Example based on the one given by Wright [260].

metric it is weighted more highly. Since m is an unbounded positive real number, it is mapped into the range (0–1]. This gives us the FUM score:

$$\text{FUM score} = 4 \cdot f + 3 \cdot u + 3 \cdot \frac{2}{1 + e^m}$$

The uncertainty in f, u and m can be computed. f is computed by taking the number of non-vulnerable device days (i) and dividing it by the total number of vulnerable (v) and non-vulnerable device days ($f = i/(i + v)$). The number of non-vulnerable device days and the number of vulnerable device days are both counting experiments and so their measurement error is their square root [229]. Since the numbers involved are large, the uncertainty in f is small. The value u is computed by taking the sum of the proportions of devices running the most recent version each day, both the count of devices running the maximum version and total count have square root uncertainties. The value m is computed by counting the number of vulnerabilities that affected that entity and that have not yet been fixed on any device observed from that entity every day and averaging over observed days. However, it could be that the entity has released a fix to some devices but a device with that fix has not yet been observed. So the uncertainty in the measurement is the probability of not having observed a fixed device if a fixed device existed. I assume that if the fix has been released then

Name	f	u	m	**score** (out of 10)
Nexus devices	0.53 ± 0.00	0.50 ± 0.00	0.69 ± 0.01	5.63 ± 0.02
non-Nexus	0.09 ± 0.00	0.02 ± 0.00	0.74 ± 0.00	2.35 ± 0.00

Table 6.2: Security scores for Nexus

at least 1.0% of devices have the fix.[1] This gives an uncertainty of 0.99^n where n is the number of devices contributing to that day's data for each vulnerability outstanding each day. The Python `uncertainties` library was used to propagate uncertainties through calculations. This does not capture systematic errors. For example, manufacturer specific vulnerabilities are not included, however performance in fixing manufacturer specific vulnerabilities should be strongly correlated with performance fixing vulnerabilities affecting all of Android.

6.3 Results: Security scores

On average, between July 2011 and March 2016, I found $12.4 \pm 0.0\%$ of device to be free from known vulnerabilities, $5.67 \pm 0.0\%$ of devices to run the most recent version of Android and 0.661 ± 0.0 outstanding vulnerabilities not fixed on any device. This gives a security score of 2.71 ± 0.0 out of 10. However, there are a wide variety of scores depending on the source of the device. There is anecdotal evidence that Google's Nexus devices are better at getting updates than other Android devices because Google makes the original updates and ships them to its devices [135]. Table 6.2 shows that this is indeed the case with Nexus devices getting much better scores than non-Nexus devices. Different device manufacturers have very different scores, Table 6.3 shows the scores for the 11 device manufacturers with a significant presence in the data with *LG* (4.28 ± 0.0 out of 10) scoring highest and *walton* (0.284 ± 0.008 out of 10) scoring lowest. Device manufacturers are considered significant if there is data from at least 100 devices and at least 10 000 days of contributions. Additionally, for m and u days with less than 20 devices contributing to that day's score are ignored.

Even within device manufacturers, different models can have very different update behaviours and hence security. Table 6.4 shows the results for the 20

[1]The selection of 1.0% is arbitrary, but if a version has only been deployed to a smaller proportion then it does not have a substantial penetration.

Name	f	u	m	**score** (out of 10)
LG	0.34 ± 0.00	0.33 ± 0.00	0.74 ± 0.01	4.28 ± 0.02
Motorola	0.26 ± 0.00	0.14 ± 0.00	0.65 ± 0.01	3.50 ± 0.02
HTC	0.13 ± 0.00	0.09 ± 0.00	0.87 ± 0.01	2.59 ± 0.02
Sony	0.13 ± 0.00	0.18 ± 0.00	1.09 ± 0.02	2.57 ± 0.02
Asus	0.23 ± 0.00	0.46 ± 0.01	5.61 ± 0.06	2.29 ± 0.02
Samsung	0.11 ± 0.00	0.06 ± 0.00	0.99 ± 0.00	2.24 ± 0.00
other	0.04 ± 0.00	0.05 ± 0.00	1.14 ± 0.01	1.79 ± 0.02
oneplus	0.02 ± 0.00	0.31 ± 0.01	7.85 ± 0.13	1.00 ± 0.02
alps	0.02 ± 0.00	0.20 ± 0.01	5.00 ± 0.10	0.73 ± 0.02
Symphony	0.00 ± 0.00	0.10 ± 0.00	5.74 ± 0.06	0.32 ± 0.01
walton	0.00 ± 0.00	0.09 ± 0.00	6.08 ± 0.08	0.28 ± 0.01

Table 6.3: Security scores for manufacturers

Made by	Model name	f	u	m (out of 10)	**score**
LG	Nexus 5	0.71	0.65 ± 0.01	6.09 ± 0.09	4.80 ± 0.03
Samsung	Galaxy Nexus	0.51	0.53 ± 0.01	1.50 ± 0.04	4.74 ± 0.05
Asus	Nexus 7	0.34	0.67 ± 0.01	5.69 ± 0.08	3.39 ± 0.03
LG	Nexus 4	0.36	0.57 ± 0.01	5.52 ± 0.08	3.21 ± 0.03
HTC	HTC Desire	0.14	0.07 ± 0.01	0.51 ± 0.03	3.03 ± 0.05
HTC	Desire HD	0.09	0.05 ± 0.00	0.43 ± 0.03	2.86 ± 0.04
unknown	*other*	0.08	0.13 ± 0.00	0.74 ± 0.00	2.65 ± 0.00
Samsung	GT-I9000	0.03	0.03 ± 0.00	0.44 ± 0.04	2.55 ± 0.05
HTC	HTC Sensation	0.34	0.01 ± 0.01	1.49 ± 0.06	2.49 ± 0.06
Motorola	DROIDX	0.02	0.04 ± 0.01	0.55 ± 0.04	2.39 ± 0.06
Samsung	GT-I9100	0.21	0.01 ± 0.00	1.26 ± 0.02	2.23 ± 0.02
HTC	HTC Desire S	0.02	0.02 ± 0.00	1.00 ± 0.08	1.75 ± 0.09
HTC	HTC One	0.10	0.38 ± 0.01	6.08 ± 0.12	1.56 ± 0.04
Samsung	GT-N7000	0.25	0.00 ± 0.00	2.46 ± 0.05	1.46 ± 0.03
Samsung	GT-P1000	0.01	0.00 ± 0.01	1.87 ± 0.06	0.84 ± 0.05
Samsung	GT-I9505	0.07	0.13 ± 0.00	7.03 ± 0.07	0.70 ± 0.01
Samsung	SM-N9005	0.05	0.15 ± 0.01	8.65 ± 0.20	0.65 ± 0.03
Samsung	GT-I9300	0.14	0.02 ± 0.00	6.46 ± 0.05	0.62 ± 0.01
HTC	HTC Desire HD	0.00	0.00 ± 0.01	3.03 ± 0.05	0.28 ± 0.03
Samsung	GT-N7100	0.05	0.00 ± 0.01	6.64 ± 0.09	0.21 ± 0.02
Symphony	Symphony W68	0.00	0.00 ± 0.01	11.00 ± 0.13	0.00 ± 0.03

Table 6.4: Security scores for models. Values for f are all ± 0.00.

Name	f	u	m	**score** (out of 10)
O2 uk	0.27 ± 0.00	0.12 ± 0.00	0.30 ± 0.01	3.97 ± 0.02
Sprint	0.19 ± 0.00	0.12 ± 0.00	0.30 ± 0.01	3.67 ± 0.02
T-Mobile	0.21 ± 0.00	0.17 ± 0.00	0.56 ± 0.02	3.56 ± 0.03
3	0.16 ± 0.00	0.11 ± 0.00	0.36 ± 0.02	3.44 ± 0.03
Orange	0.20 ± 0.00	0.11 ± 0.00	0.53 ± 0.03	3.35 ± 0.04
AT&T	0.12 ± 0.00	0.09 ± 0.00	0.37 ± 0.01	3.20 ± 0.02
Vodafone uk	0.11 ± 0.00	0.14 ± 0.00	0.54 ± 0.03	3.08 ± 0.04
Verizon	0.16 ± 0.00	0.08 ± 0.00	0.58 ± 0.01	3.04 ± 0.02
unknown	0.12 ± 0.00	0.19 ± 0.00	1.01 ± 0.02	2.66 ± 0.02
n Telenor	0.06 ± 0.00	0.13 ± 0.00	1.31 ± 0.02	1.93 ± 0.02
Airtel	0.05 ± 0.00	0.04 ± 0.00	2.48 ± 0.04	0.76 ± 0.02
Robi	0.00 ± 0.00	0.09 ± 0.00	3.14 ± 0.05	0.51 ± 0.02
Grameenphone	0.00 ± 0.00	0.04 ± 0.00	3.08 ± 0.03	0.40 ± 0.01
banglalink	0.00 ± 0.00	0.04 ± 0.00	3.12 ± 0.05	0.37 ± 0.01

Table 6.5: Security scores for operators

device models that have a significant presence by the same metric, with *Nexus 5* (4.8 ± 0.0 out of 10) scoring highest and *Symphony W68* (0.0001 ± 0.0283 out of 10) scoring lowest. To test whether this seems fair the version data for the highest and lowest scoring models can be compared. Figure 6.3c shows the full version distribution for *Symphony W68*, which is only observed running one version. Figure 6.3b shows the full version distribution for *HTC Desire HD A9191*, which is the third worst model and for which there is more historical data; it shows it received one update at the beginning of 2012, which was deployed fairly rapidly to most devices, but received no further updates. Figure 6.3a shows the same information for *Galaxy Nexus* (the second highest ranked model), which received 49 different versions, some of which were only deployed to small number of devices, but the distribution for all devices regularly and rapidly transitions from one version to another before ending up on '4.3 JWR66Y'. Both *Galaxy Nexus* and *HTC Desire HD A9191* device models start off with the full version string of '2.3.3 GRI40' but the *Galaxy Nexus* receives many more updates over the same time period. Other models from the same manufacturer with similar model names to *HTC Desire HD A9191* do much better such as the *Desire HD*.

I also analysed the 18 network operators with a significant presence in the

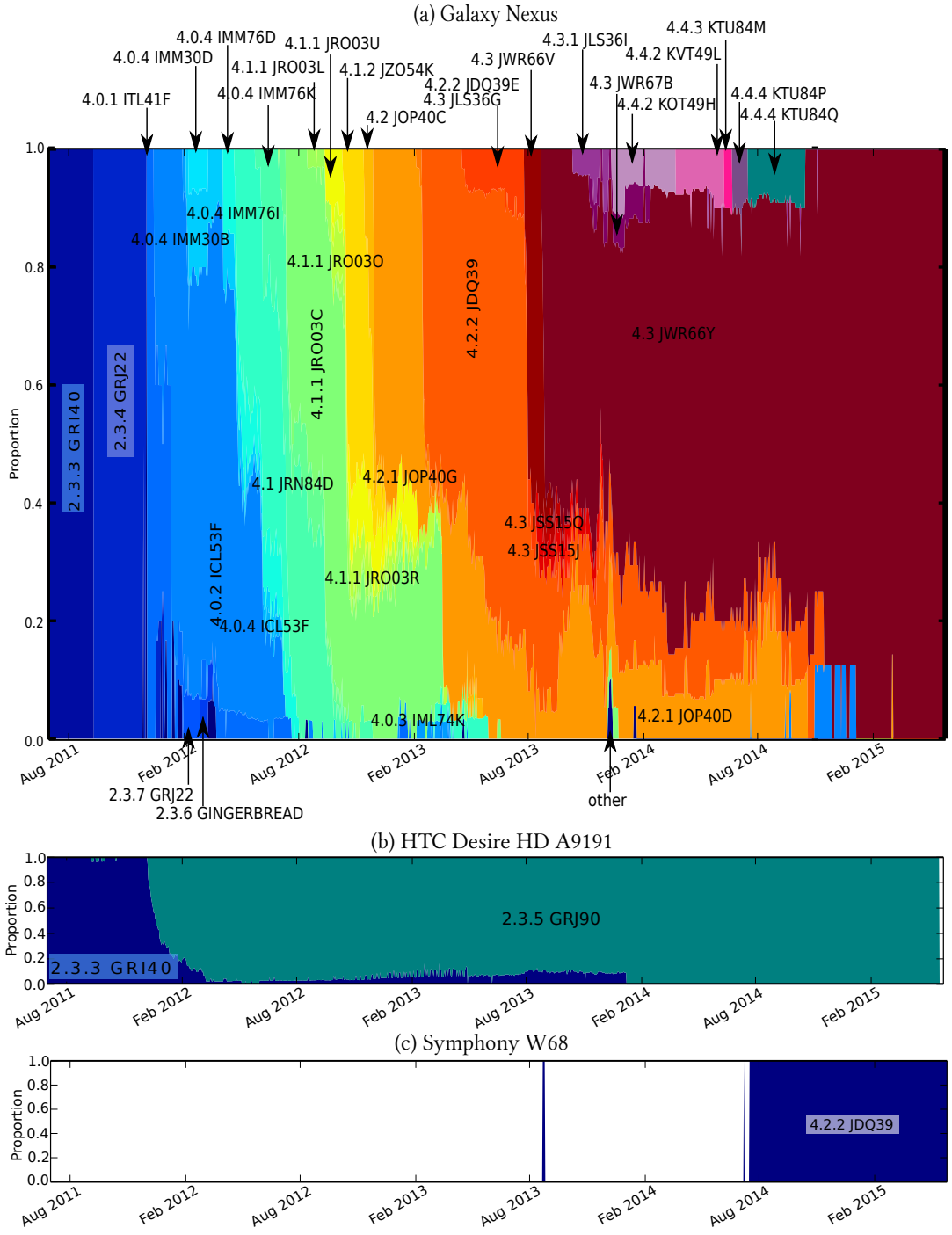


Figure 6.3: Full version distributions for the highest (a) and lowest (b,c) scoring models.

data. Table 6.5 shows the results with *O2 uk* (3.97 ± 0.0 out of 10) scoring highest and *banglalink* (0.366 ± 0.013 out of 10) scoring lowest. However, the score of a network operator is affected by the manufacturers of the devices that are in use on its network. This is in turn affected by both the device models a network operator offers to users and upon user's choice of device models. Hence, having a worse score does not necessarily mean that a network operator is worse, it could be that its users all pick devices from a worse device manufacturer, for example, because those devices were cheaper. A network operator could use data from this chapter to exclude vulnerable devices from those offered to consumers. An added value analysis of network operators, that takes into account the device mix used by users of that network operator, would make it possible to determine whether a network operator is making the situation better or worse by the way it ships updates to users. However, the sample size is is too small to do that because while there is a significant numbers of devices for each of the 20 device models (Table 6.4) and for each of the 18 network operators (Table 6.5), a significant number of each model in each network operator would be required. Since the distribution of devices is unlikely to be uniformly distributed across device models and network operators an estimated 100 000 unique devices would be required each day for at least a year. This much data could be collected, but they are two orders of magnitude more than is available from the Device Analyzer data set.

6.4 Update bottleneck

If update delays are caused by manufacturers rather than operators or users, one would expect the update behaviour of devices with the same device model to be similar and rapid. I found that within 14 days of the first observation of a new version on a device, half of all devices of that model have the new version (or a higher version) installed, and within 252 days 95% of devices have the new version (or a higher version). This compares with the average rates of deployment for Android OS versions of 350 days for half and 1 100 days for 95%. There is a variation between device models, with the update distributed to most devices quickly and others having a much slower roll out, but since some device models do update quickly the bottleneck is unlikely to be with the

user. Perhaps some device models are preferred by users who are more likely to install updates than others, however I do observe updates rolled out to some device models quickly and user behaviour is not beyond the control of the device manufacturer. They could install updates automatically or pester the user into installing them, and at least some of them do pester. Silent automatic updates have been shown to boost uptake [87].

6.5 Scores over time

The scoring metric as originally computed, is averaged over the whole history of the device manufacturer, device model or network operator. It gives equal weight to both periods years ago and to the last few months. If instead an exponential moving average of the daily score is used, for days with more than 20 devices and when there have been at least 100 consecutive days of data with that many devices, the score can be plotted over time. Equation 6.1 shows how the value for a particular day (v_i) is computed from the previous day's value and the input for the current day (n) with an α of $1/100$.

$$v_i = v_{i-1}(1 - \alpha) + n\alpha \tag{6.1}$$

Figure 6.4 shows this for manufacturers, Figure 6.5 for device models, Figure 6.6 for network operators and Figure 6.7 for Nexus and non-Nexus devices. On these figures the 95% confidence intervals are indicated. These show how the scores for different entities are different and change over time, while there is correlated behaviour for different entities, for instance, when new vulnerabilities affecting all Android are discovered, these lines still have crossings due to the different behaviour of the various entities. It also shows that there is insufficient data for some of the entities some of the time, resulting in gaps in the data. The clearest results are for Figure 6.7 with a large gap between the scores for Nexus and non-Nexus devices across the whole data set.

6.6 Sensitivity of scoring metric

The Spearman's Rank correlation coefficient is used to evaluate whether the ranking of different manufacturers is sensitive to the form of the scoring met-

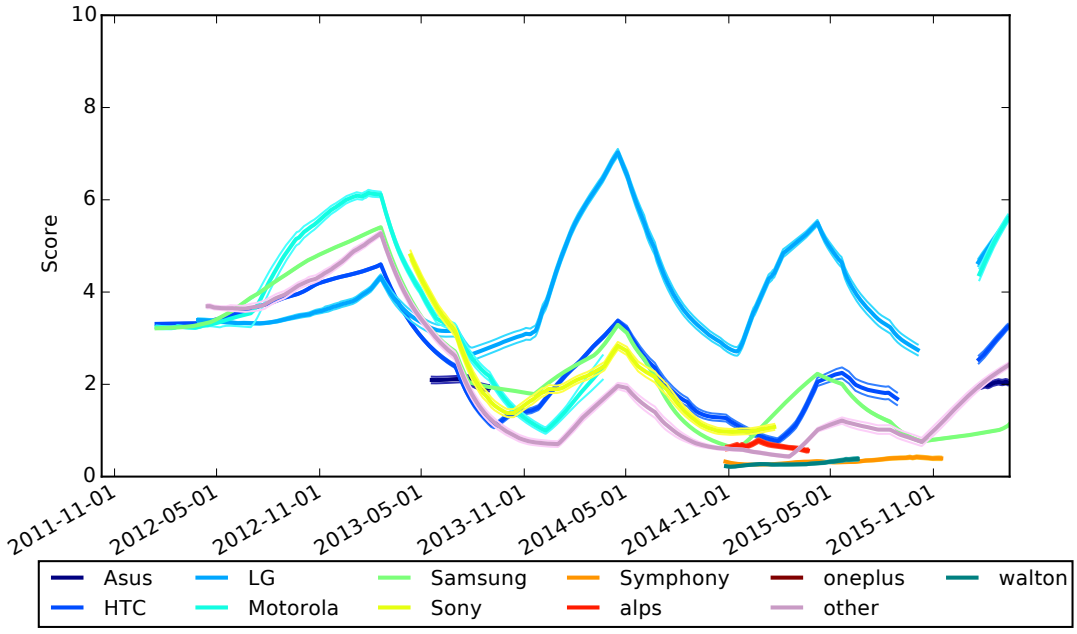


Figure 6.4: Security scores for device manufacturers

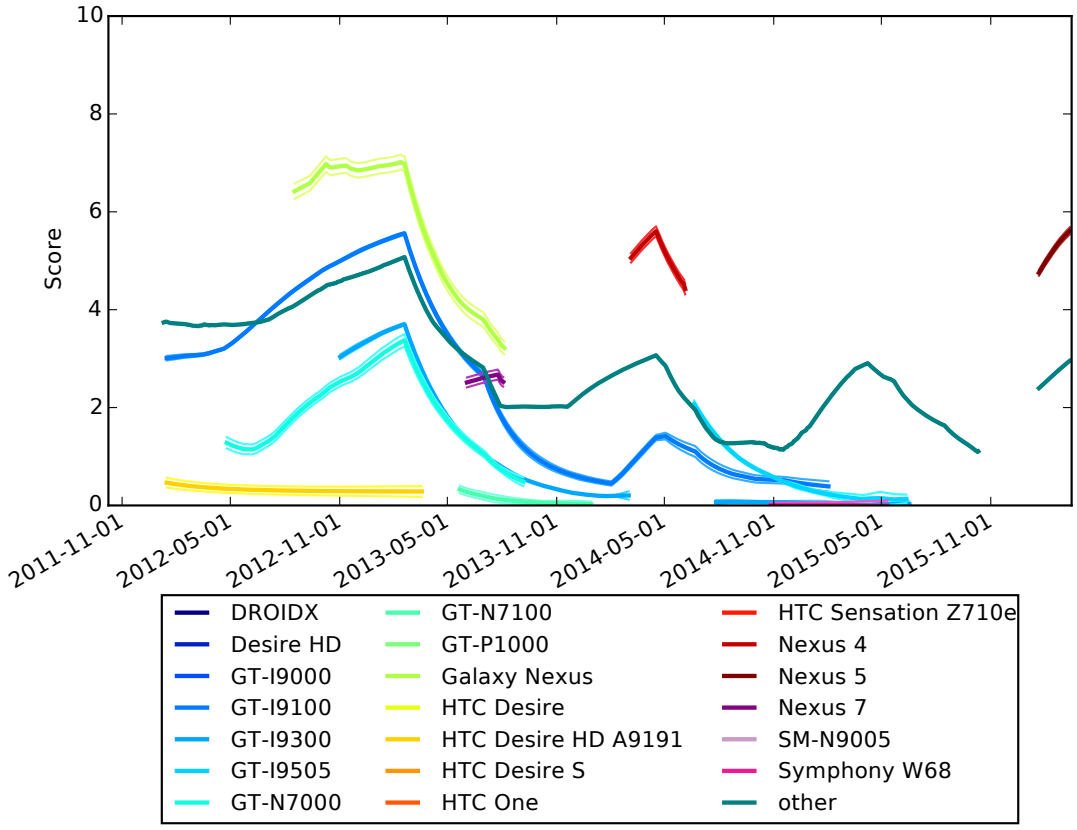


Figure 6.5: Security scores for device models

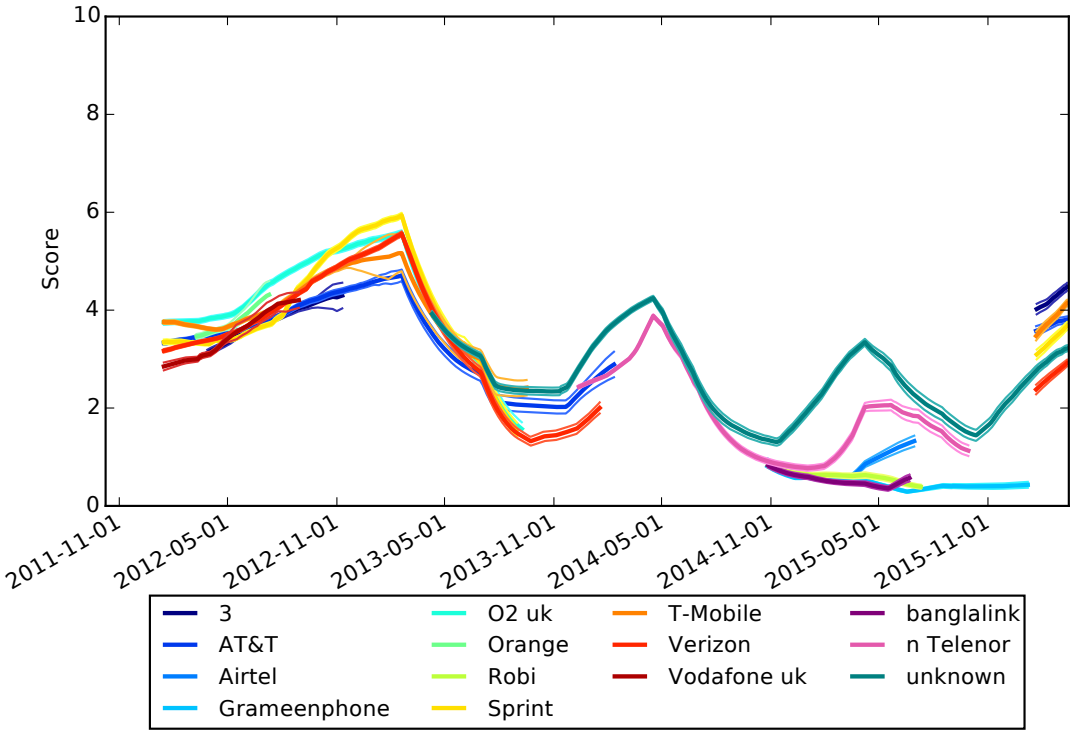


Figure 6.6: Security scores for network operators

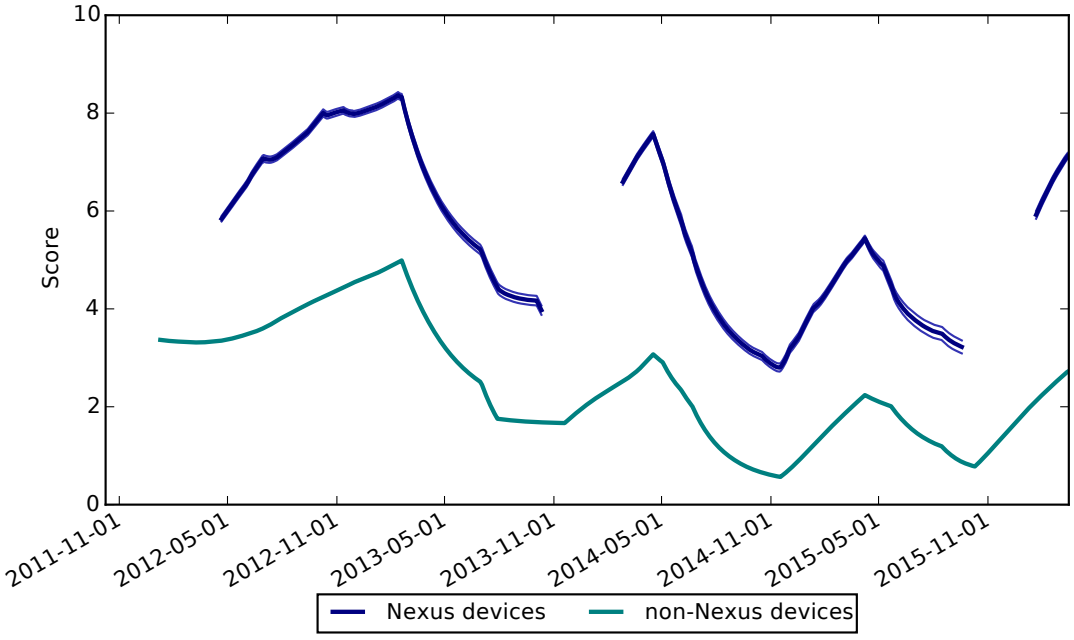


Figure 6.7: Security scores for Nexus and non-Nexus devices

	manufacturer	model	operator	Nexus
$\pm\sigma$	0.2	0.141	0.175	0.632
u	0.3	0.712	0.552	1.0
m	0.855	0.517	0.934	1.0
f	0.964	0.694	0.943	1.0
weight m	0.973	0.948	0.991	1.0
equal	0.991	0.996	1.0	1.0
weight u	0.991	0.996	1.0	1.0

Table 6.6: Spearman Rank correlation coefficients for different metrics. The uncertainty is constant for each column but does not take into account the uncertainty in the score which produced the ranking.

	manufacturer	model	operator	Nexus
m	-1.21 ± 2.08	-0.819 ± 2.93	-3.03 ± 1.84	-2.58 ± 2.2
weight m	-0.201 ± 0.249	-0.116 ± 0.412	-0.426 ± 0.25	-0.347 ± 0.314
equal	-0.0894 ± 0.0513	-0.0377 ± 0.147	-0.137 ± 0.077	-0.0991 ± 0.105
weight u	-0.0671 ± 0.128	0.00267 ± 0.17	0.0162 ± 0.0816	0.0494 ± 0.0373
u	0.134 ± 1.58	0.366 ± 1.78	1.39 ± 1.11	1.39 ± 1.08
f	0.804 ± 0.462	0.34 ± 1.21	1.23 ± 0.61	0.892 ± 0.84

Table 6.7: Mean change in scores for different metrics

ric. This compared between lists of manufacturers etc. sorted according to different forms of the scoring metric, this is shown in Table 6.6. In the table, the 'equal' metric weights f, u and m equally rather than favouring f and makes little difference. Similarly weighting u or m more highly rather than f makes little difference. While the f, u and m components do have some correlation with the overall FUM score, the rankings produced vary substantially, showing that the composite FUM score cannot be replaced by one of its component parts. Changing the scoring metric also impacts the scores given for each entity Table 6.7 shows the mean impact on the scores. This shows that m tends to drag down scores.

6.7 Utilitarianism

From a utilitarian standpoint, while small manufacturers like Symphony and Walton do badly on the scores, they do not have as many customers as higher scoring manufacturers. Hence the total risk to users from the higher scoring popular manufacturers is higher than the risk from the lower scoring unpopular

manufacturers. We could normalise for market penetration and so give a score reflecting the risk posed by that manufacturer's performance, which would tend to decrease the difference between manufacturers in the current scoring. Since the scores are provided so that customers can chose which devices to buy then the marginal risk to an individual is of interest rather than the aggregate risk to all users.

6.8 Gaming the score

If the comparative data given here is used to influence purchasing decisions then entities in the Android ecosystem might try to game the score rather than genuinely improve security. The value of f is hard to game without doing a good job at security but it does not get any worse if there is already one known vulnerability and another is found. A high value of u could be achieved by only ever shipping one version but that would give low values for f and m (and not be attractive to new customers). A high value of m could be achieved by focusing on only one device at a time and ensuring that it gets updates but ignoring all others, but that would lower f and u. One way to influence the scores would be to add additional devices to Device Analyzer, that have good security, these would have to be real end user devices since we could detect fake ones if they deviated from the behaviour of real devices in Device Analyzer. This would increase the size of the data set and would require providing genuinely good security to some users. Some active attacks like blocking access to the Device Analyzer servers from the mobile data network would not be effective as Device Analyzer would retry on Wi-Fi. Other denial of service attacks on the Device Analyzer servers might be effective but illegal. Some entities might be able to force the uninstallation of the app from all devices. Therefore, the score is secure against passive gaming attacks that change the measured distribution, but would require active defence against active gaming attacks, which target the measurement devices.

6.9 Summary

The security of Android depends on the timely delivery of updates to fix critical vulnerabilities. Unfortunately few devices receive prompt updates, with an

overall average of 1.43 updates per year, leaving devices unpatched for long periods. The bottleneck for the delivery of updates in the Android ecosystem rests with the manufacturers, who fail to provide updates to fix critical vulnerabilities. This arises in part because the market for Android security today is like the market for lemons: there is information asymmetry between the manufacturer, who knows whether the device is currently non-vulnerable and will receive updates, and the consumer, who does not.

Consequently there is little incentive for manufacturers to provide updates. To address this issue the FUM security metric quantifies and ranks the performance of device manufacturers and network operators, based on their provision of updates and exposure to critical vulnerabilities. This metric enables purchasers and regulators to determine which device manufacturers and network operators provide updates and which do not.

The corpus of 24 600 devices from Device Analyzer demonstrates that there is significant variability in the timely delivery of security updates across different device manufacturers and network operators. The Android ecosystem as a whole gets a FUM security score of 2.71 out of 10. In the Device Analyzer data, Nexus devices do considerably better than average with a score of 5.63; and LG is the best manufacturer with a score of 4.28.

By quantifying the Android update process and providing concrete numbers on the flow of updates and their latency and through analysis of the deployment of updates to device models this chapter showed that the main update bottleneck lies with manufacturers rather than Google, operators or users.

The FUM metric could also be applied to other software ecosystems and be used to evaluate the security of other operating systems or applications. It only requires the availability of data on the distribution deployed versions of the software over time, and the vulnerabilities that affected those versions.

CHAPTER 7

FUTURE WORK

7.1 Direct extensions

There are several ways in which the work presented in this dissertation could be directly extended. Device Analyzer tracks the versions of apps which are installed and so it would be possible to investigate the upgrade patterns of Android apps and compare them with the upgrade patterns of the Android OS. Similarly, some Android apps are manufacturer specific and not distributed through Google Play (some vulnerabilities in them compromise the security of the device), do they follow the upgrade patterns of apps in Google Play or of the OS?

Additional data on Android versions and vulnerabilities would allow more accurate and representative results and larger quantities would allow comparisons to be made between more manufacturers and devices.

Device Analyzer is only available on Google Play, not on any alternative markets, perhaps they have different behaviours?

The FUM security metric could be applied to other platforms such as iOS and Windows if suitable sources of version data and vulnerability data could be found. This would then allow a direct comparison to be made between the security of different platforms.

7.2 New approaches

The FUM security metric does not fully capture the security of a computer system or ecosystem. The development of other metrics to evaluate security beyond vulnerability to malicious code could address this. For example, the FUM security metric does not account for the impact of remote code execution vulnerabili-

ties compared with local privilege escalation vulnerabilities, or evaluate physical security (resistance to theft), privacy preservation or phishing prevention.

With good data and metrics to measure the security of computer systems between which a consumer can choose, the next step is to present it to the consumer. Conveying that information in a useful way that allows them to easily make trade-offs between different products based on that information would require further work.

With data and metrics on the security of computer systems, using them to inform regulators, so that they can make good decisions and understand the impact of their regulation, is likely to be complex. Perhaps the metrics and presentation mechanisms that are useful for regulators might be different from those that are most useful for consumers.

If security updates are made available promptly, then getting users to install them might be the next bottleneck, and so understanding the best methods for encouraging users to install them would be useful. There are technical measures, such as making updates easier to install, or installing updates automatically, which have been shown to help. However, there are also human aspects to this, such as how users are told about the update and the importance of installing it. There have already been some interview and log data studies looking at Windows updates and updates to apps [250], which have identified some of the reasons users choose not to upgrade [241]. Different strategies for notifying users of updates have not yet been evaluated, particularly for users who are sysadmins.

CHAPTER 8

CONCLUSION

The plural of anecdote is not data.
Roger Brinner

There has been a steady discovery of critical vulnerabilities in Android and, hence, a need to quickly deploy security updates to devices. The latency of security updates means that the Android sandbox is ineffective in the majority of cases with an average of $87.6 \pm 0.0\%$ of Android devices exposed to known critical vulnerabilities, which allow a malicious app to break out of the sandbox. Only $5.67 \pm 0.0\%$ of devices run the latest version of Android, and devices apply 1.43 ± 0.01 updates each year, less than the critical vulnerability discovery rate of between 3.79 ± 0.84 and 7.96 ± 1.23. Despite this high level of vulnerability, Android does not suffer from substantial malware problems. This indicates that the Google Play Store and Verify Apps protect devices from malicious apps.

However, protection from the store and at install time is not enough: Chapter 5 demonstrated that remote code execution is practical and bypasses both the Google Play Store and Verify Apps. The exponential decay model was shown to be a good fit for Android API vulnerabilities and applying this model to this JavaScript-to-Java interface vulnerability showed that, for apps that are vulnerable only on outdated devices, 95% of all Android devices will be protected by August 2017, 4.82 ± 0.97 years after the release of the fix.

In the long term, there needs to be greater emphasis on providing timely updates. This could be fixed through a variety of mechanisms. Increased regulation is one potential solution. Currently, manufacturers do not even provide updates for the length of the contracts under which the users originally bought the phone, despite efforts by the ACLU to persuade the FTC to force them to do so [215], and despite the FTC forcing HTC to do so [109].

The fundamental problem is one of a misalignment of incentives in the current Android ecosystem: there are too few reasons why manufacturers should provide timely security updates. A more rigid vertically-integrated market would likely improve the security of Android at the cost of some of the freedom it currently affords manufacturers and users – the very thing that gained it a majority market share.

Alternatively, decentralised management might work. The claim of the open-source software 'bazaar' is that it is more secure because anyone can review the source code and provide fixes [199]. On the surface, Android is an open-source project, and therefore others should be able to step in when the manufacturer fails to provide timely updates. Unfortunately, only manufacturers can provide updates today because the complete source code for the (often modified) operating system, as well as the drivers and the build environment, including the signing keys, are not available for many devices (§6.1). Third parties also cannot ship updates so that they appear for users to install directly. Users have to hunt for a custom ROM. While Cyanogenmod, an open-source community supported version of Android, has supported 368 device models, there are 3 720 device models in Device Analyzer. This is unlikely to change since there are economic disincentives from the manufacturers' perspective, and, in any case, there is a trust issue that needs to be solved: how does the user determine whether a binary from an alternative supplier is more secure than the one they already have installed?

To address the issue of updates not being provided, I developed the FUM security metric to quantify and rank the performance of device manufacturers and network operators, based on their provision of updates and exposure to critical vulnerabilities. The metric enables purchasers and regulators to determine which device manufacturers and network operators provide updates and which do not.

Using a corpus of 24 600 devices I demonstrated that there is significant variability in the timely delivery of security updates across different device manufacturers and network operators. Across the ecosystem as a whole I assign a FUM security score of 2.71 out of 10. In the Device Analyzer data, Nexus devices do considerably better than average with a score of 5.63; LG is the best manufacturer with a score of 4.28.

Installing regular updates is the best defence against malware. However, as Chapter 4 showed, these updates are rarely forthcoming. Therefore, Android

users need to take additional steps to protect themselves. The top three pieces of advice based on this dissertation are: Firstly, users should only install apps from Google Play, which has been shown to have a lower level of malware than third-party markets, and is known to perform static and dynamic analysis of available apps. Secondly, users should enable the Google Verify Apps feature if available (this is the default). Thirdly, users should avoid using untrustworthy Wi-Fi networks or use them in conjunction with a VPN tunnel to a trusted network, in order to avoid attacks from the local network (Chapter 5). However, it is unreasonable to expect users to follow this third piece of advice without expert assistance.

While there are problems with Android security, Google has taken a pragmatic three-pronged approach to improving the security of Android. Firstly, since Google controls the Google Play Store, the main entry route for code reaching Android devices (in Europe and North America), Google can use its Bouncer system to detect and remove malicious apps that have been uploaded [124]. Google also imposes an economic barrier on developers of a $25 fee for registration [122], which the developer will lose if found to be malicious. To protect devices obtaining apps from sources other than Google Play, Google have deployed the Verify Apps feature, which scans the app binary and checks that it is not malicious both before installation and periodically after installation [124]. Secondly, to address the problem of lack of updates, Google have increasingly moved core OS components away from only updating through full OS updates, to be apps that can be updated through the Google Play Store or into Google Play Services which updates through the store [198]. For example, in Android 5.0, Google made the WebView component (a vulnerability in which is discussed in Chapter 5) updatable via the Google Play Store [124]. Google have also tried to encourage manufacturers to ship updates, such as committing Motorola to shipping updates for the Moto G. Thirdly, Google have implemented technologies in Android to mitigate vulnerabilities such as SEAndroid [213], which is included in Android from version 4.1 [123], and fully enforcing from version 5.0 [124], which stops some classes of vulnerabilities from being exploitable.

My approach of measuring how vulnerabilities affect the security of a software ecosystem relies on the availability of longitudinal data about the versions of software running on devices and the vulnerabilities found in different versions of the software. Measuring security more generally relies on the availability of

representative comparable longitudinal data on the various aspects of the security measured. For example: measuring resistance to phishing requires data on which phishing attempts were successful; which failed; and why. Resistance to physical compromise requires data on the number of instances of physical compromise was attempted and the success rate. Resistance to privacy compromise requires data on what data has been accessed, and whether collection and use violates the expected privacy of the user. Unfortunately, such data is not generally available.

The collection of representative longitudinal data sets about the behaviour of computer systems and their users is an important task for the research community. Without it, people may claim things to be true and recount anecdotes that support their hypotheses but they cannot be scientific. Without a scientific understanding of the behaviour of computer systems and their users, markets cannot be optimal as they lack information. Following on from this, regulators cannot effectively regulate without knowing what is really happening or what effect their regulations have. Failure to recognise the collection of data and the production of tools to collect it as valuable contributions is a serious hindrance to research.

Scaling this data collection to the measurement of the IoT will be a difficult challenge and not something that one individual or institution can accomplish alone. While methods like those I have used to evaluate the security of Android would be applicable to IoT devices, these methods rely on the collection of data like that provided by Device Analyzer and AndroidVulnerabilities.org, both of which are expensive and time-consuming. It is important to develop methods of scaling data collection to the IoT while protecting the personal privacy and corporate secrecy of the contributors.

The Internet of Things might provide opportunities for improved efficiency, quality of service, and perhaps even security. However, it is not clear that it will be secure, or able to put the interests of the individual first. I, for one, am not going to rush out and buy Internet connected-light switches just yet.

BIBLIOGRAPHY

[1] *A short history of the BlackBerry*. 2012. URL: https://web.archive.org/web/20120302021534/http://www.bbscnw.com/a-short-history-of-the-blackberry.php — https://archive.is/lkx6H (visited on 2016-04-06) (cit. on p. 30).

[2] Mustafa Acer and Collin Jackson. "Critical vulnerability in browser security metrics". In: *Web 2.0 Security & Privacy (W2SP)* (2010) (cit. on p. 97).

[3] Anne Adams and Martina Angela Sasse. "Users are not the enemy". In: *Communications of the ACM* 42.12 (1999), pp. 40–46. DOI: 10.1145/322796.322806 (cit. on p. 36).

[4] David Ahmad. "Two Years of Broken Crypto: Debian's Dress Rehearsal for a Global PKI Compromise". In: *IEEE Security & Privacy Magazine* 6.5 (2008), pp. 70–73. ISSN: 15407993. DOI: 10.1109/MSP.2008.131 (cit. on p. 25).

[5] George A. Akerlof. "The market for "lemons": Quality uncertainty and the market mechanism". In: *The Quarterly Journal of Economics* 84.3 (1970), pp. 488–500 (cit. on p. 16).

[6] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Cranor, and Yuvraj Agarwal. "Your Location has been Shared 5,398 Times! A Field Study on Mobile App Privacy Nudging". In: *Proc. of the 2015 ACM conference on Human factors in computing systems (CHI)*. Seoul, Republic of Korea: ACM, 2015-04, pp. 787–796. ISBN: 9781450331456. DOI: 10.1145/2702123.2702210 (cit. on p. 37).

[7] Jonathan Anderson, Joseph Bonneau, and Frank Stajano. "Inglorious Installers: Security in the Application Marketplace." In: *WEIS* (2010), pp. 0–44 (cit. on p. 42).

[8] Nate Anderson. *Confirmed: US and Israel created Stuxnet, lost control of it*. 2012-06. URL: http://arstechnica.com/tech-policy/2012/06/confirmed-us-israel-

created-stuxnet-lost-control-of-it/ — https://archive.is/hfe4k (visited on 2015-09-08) (cit. on p. 23).

[9] Ross Anderson. "API Attacks". In: *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2008, pp. 547–558. ISBN: 978-0-470-06852-6 (cit. on p. 26).

[10] Ross Anderson. "Distributed Systems". In: *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd ed. Wiley, 2008, pp. 115–133. ISBN: 978-0-470-06852-6 (cit. on p. 23).

[11] Ross Anderson. "Managing the Development of Secure Systems". In: *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd ed. Wiley, 2008, pp. 815–856. ISBN: 978-0-470-06852-6 (cit. on p. 36).

[12] Tim Anderson. *Microsoft to Windows 10 consumers: You'll get updates LIKE IT or NOT*. 2015-07. URL: http://m.theregister.co.uk/2015/07/16/windows_10_will_update_whether_you_like_it_or_not_unless_you_have_enterprise_edition/ — https://archive.is/T3Os9 (visited on 2015-07-20) (cit. on p. 15).

[13] Android. *Codenames, Tags, and Build Numbers*. 2016. URL: https://source.android.com/source/build-numbers.html — https://archive.is/DTqYH (visited on 2016-03-02) (cit. on p. 61).

[14] Jeremy Andrus, Christoffer Dall, Alexander Van Hof, Oren Laadan, and Jason Nieh. "Cells : A Virtual Mobile Smartphone Architecture". In: *Symposium on Operating System Principles (SOSP)* (2011), pp. 173–187. DOI: 10.1145/2043556.2043574 (cit. on p. 47).

[15] *Appannie Google Play analysis*. URL: https://archive.is/PRPNn — https://web.archive.org/web/20150113230539/http://www.appannie.com/search/?vertical=apps&market=google-play (cit. on p. 15).

[16] *Apps available from leading app stores*. URL: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ — https://archive.is/rQQ4k (visited on 2015-09-25) (cit. on pp. 13, 15).

[17] Ashish Arora, Rahul Telang, and Hao Xu. "Optimal Policy for Software Vulnerability Disclosure". In: *Management Science* 54.4 (2008), pp. 642–656. ISSN: 0025-1909. DOI: 10.1287/mnsc.1070.0771 (cit. on p. 41).

[18] Daniel Arp, Michael Spreitzenbarth, Hubner Malte, Hugo Gascon, and Konrad Rieck. “Drebin: Effective and Explainable Detection of Android Malware in Your Pocket”. In: *Network and Distributed System Security (NDSS)*. San Diego, CA, USA: Internet Society, 2014-02, pp. 23–26. ISBN: 1891562355 (cit. on p. 45).

[19] Gregory V. Bard. “Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric”. In: *Conferences in Research and Practice in Information Technology Series* 68 (2007), pp. 117–124. ISSN: 14451336 (cit. on p. 93).

[20] Adam Barth, Saung Li, Benjamin I. P. Rubinstein, and Dawn Song. *How Open Should Open Source Be?* Tech. rep. 2011-09. arXiv: arXiv:1109.0507v1 (cit. on p. 41).

[21] Mihir Bellare, A. Desai, E. Jokipii, and P. Rogaway. “A concrete security treatment of symmetric encryption”. In: *Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc, 1997-10, pp. 394–403. ISBN: 0-8186-8197-7. DOI: 10.1109/SFCS.1997.646128 (cit. on p. 28).

[22] Anthony Bellissimo, John Burgess, and Kevin Fu. “Secure software updates: disappointments and new challenges”. In: *USENIX Hot Topics in Security*. USENIX, 2006, pp. 37–43 (cit. on p. 42).

[23] Alastair R. Beresford and Frank Stajano. “Location privacy in pervasive computing”. In: *IEEE Pervasive Computing* 2.1 (2003), pp. 46–55. ISSN: 15361268. DOI: 10.1109/MPRV.2003.1186725 (cit. on p. 37).

[24] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. “MockDroid: trading privacy for application functionality on smartphones”. In: *Proceedings of the 11th Workshop on Mobile Computing Systems and Applications HotMobile*. ACM, 2011. ISBN: 9781450306492. DOI: 10.1145/2184489.2184500 (cit. on p. 38).

[25] Neil Bergman. *Abusing WebView JavaScript Bridges*. 2012-12. URL: http://d3adend.org/blog/?p=314 — https://archive.is/MkvZF (visited on 2015-01-09) (cit. on p. 86).

[26] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. “The security impact of a new cryptographic library”. In: *LatinCrypt*. 2012, pp. 159–176. DOI: 10.1007/978-3-642-33481-8_9 (cit. on pp. 26, 29).

[27] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *IEEE Symposium on Security & Privacy*. 2015 (cit. on p. 41).

[28] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. "What the App is That? Deception and Countermeasures in the Android User Interface". In: *IEEE Symposium on Security and Privacy* (2015) (cit. on p. 39).

[29] Leyla Bilge and Tudor Dumitras. "Before We Knew It: an Empirical Study of Zero-Day Attacks in the Real World". In: *ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 833–844. DOI: 10.1145/2382196.2382284 (cit. on pp. 40, 55).

[30] BlackBerry. *BlackBerry Reports Fiscal 2016 Second Quarter Results*. 2015-09. URL: http://press.blackberry.com/content/dam/bbCompany/Desktop/Global/PDF/Investors/Documents/2016/Q2FY16_Earnings_Press_Release_FINAL-09.25.15.pdf (visited on 2016-04-06) (cit. on p. 30).

[31] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. "An Android application sandbox system for suspicious software detection". In: *IEEE International Conference on Malicious and Unwanted Software (MALWARE)* (2010), pp. 55–62. DOI: 10.1109/MALWARE.2010.5665792 (cit. on p. 45).

[32] Adrian Blomfield. *Syria 'tortures activists to access their Facebook pages'*. 2011-05. URL: http://www.telegraph.co.uk/news/worldnews/middleeast/syria/8503797/Syria-tortures-activists-to-access-their-Facebook-pages.html — https://archive.is/6Dw9d (visited on 2015-09-08) (cit. on p. 24).

[33] Joseph Bonneau. *New Facebook Photo Hacks*. URL: http://www.lightbluetouchpaper.org/2009/02/11/new-facebook-photo-hacks/ — https://archive.is/sSDej (visited on 2013-01-01) (cit. on p. 25).

[34] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes". In: *IEEE Symposium on Security and Privacy*. 2012. DOI: 10.1109/SP.2012.44 (cit. on pp. 24, 43).

[35] Marie Brewis. *Windows 10 for phones UK release date, price and new features: When will my phone get Windows 10 Mobile?* 2015-12. URL: http://www.pcadvisor.co.uk/new-product/mobile-phone/windows-phone-10-uk-release-date-price-new-features-mobile-3594482/ — https://archive.is/qi9gY (visited on 2016-04-07) (cit. on p. 30).

[36] Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen. "A trend analysis of exploitations". In: *IEEE Symposium on Security and Privacy*. 2001, pp. 214–229. ISBN: 0-7695-1046-9. DOI: 10.1109/SECPRI.2001.924300 (cit. on p. 40).

[37] Billy Bob Brumley and Nicola Tuveri. "Remote timing attacks are still practical". In: *Computer Security–ESORICS*. Vol. LNCS 6879. Springer Berlin Heidelberg, 2011, pp. 355–371. ISBN: 978-3-642-23821-5. DOI: 10.1007/978-3-642-23822-2_20 (cit. on p. 28).

[38] David Brumley and Dan Boneh. "Remote timing attacks are practical". In: *USENIX Security Symposium*. Washington, DC: USENIX, 2003, pp. 701–716. DOI: 10.1016/j.comnet.2005.01.010 (cit. on p. 28).

[39] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. "Automatic patch-based exploit generation is possible: Techniques and implications". In: *IEEE Symposium on Security and Privacy* (2008), pp. 143–157. ISSN: 10816011. DOI: 10.1109/SP.2008.17 (cit. on pp. 41, 56).

[40] US Census Bureau. *Historical National Population Estimates: July 1, 1900 to July 1, 1999*. 2000-06. URL: http://www.census.gov/popest/data/national/totals/pre-1980/tables/popclockest.txt — https://archive.is/6khTF (visited on 2015-09-24) (cit. on p. 149).

[41] U.S. Census Bureau. *Popclock*. 2015. URL: https://archive.is/OFczb — http://www.census.gov/popclock/ (visited on 2015-09-24) (cit. on p. 149).

[42] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: Behavior-Based Malware Detection System for Android". In: *ACM CCS workshop on Security and privacy in smartphones and mobile devices (SPSM)*. New York, New York, USA: ACM, 2011-10, p. 15. ISBN: 9781450310000. DOI: 10.1145/2046614.2046619 (cit. on p. 43).

[43] Mike Burrows, M. Abadi, and Roger M. Needham. “A Logic of Authentication”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 426.1871 (1989-12), pp. 233–271. ISSN: 1364-5021. DOI: 10.1098/rspa.1989.0125 (cit. on p. 29).

[44] Justin Cappos and Justin Samuel. *Package management security*. Tech. rep. University of Arizona, Computer Science Department, 2008, pp. 1–20 (cit. on p. 42).

[45] Nigel Cassidy. *Getting in a spin: Why washing machines are no longer built to last*. 2014-05. URL: http://www.bbc.co.uk/news/business-27253103 — https://archive.is/75jV9 (cit. on p. 14).

[46] CESG. *End User Devices Security Guidance: Android 4.2*. 2013-10. URL: https://www.gov.uk/government/publications/end-user-devices-security-guidance-android-42/end-user-devices-security-guidance-android-42 — https://archive.is/AwKkA (visited on 2015-07-28) (cit. on pp. 42, 55, 92).

[47] CESG. *Samsung Galaxy S6 & S6 Edge - Certification Details*. 2015-07. URL: http://www.cesg.gov.uk/servicecatalogue/Product-Assurance/CPA/Pages/Samsung-Galaxy-S6-and-S6-Edge-Certification-Details.aspx — https://archive.is/PqJyE (visited on 2015-07-24) (cit. on p. 92).

[48] Eric Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. “OAuth Demystified for Mobile Application Developers”. In: *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014-11, pp. 892–903. ISBN: 9781450329576. DOI: 10.1145/2660267.2660323 (cit. on p. 26).

[49] Xin Chen and Sencun Zhu. “DroidJust: Automated Functionality-Aware Privacy Leakage Analysis for Android Applications”. In: *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2015-06, 5:1–5:12. ISBN: 978-1-4503-3623-9. DOI: 10.1145/2766498.2766507 (cit. on p. 37).

[50] Richard Chirgwin. *BlackBerry boss kills off BB 10 OS, mulls mid-range Androids*. 2016-04. URL: http://www.theregister.co.uk/2016/04/10/blackberry_boss_mulls_midrange_mobes/ — https://archive.is/WhUYz (visited on 2016-04-12) (cit. on p. 30).

[51] Steve Christey and Barbara Pease. *An informal analysis of vendor acknowledgement of vulnerabilities*. 2001-03. URL: http://seclists.org/bugtraq/2001/Mar/154 — https://archive.today/O3Y5I (visited on 2014-06-09) (cit. on p. 41).

[52] Jeremy Clark and Paul C. van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements". In: *IEEE Symposium on Security and Privacy* (2013), pp. 511–525. DOI: 10.1109/SP.2013.41 (cit. on pp. 24, 28, 85).

[53] Richard Clayton. "Who'd Phish from the Summit of Kilimanjaro?" In: *Financial Cryptography and Data Security*. Vol. 3570. Springer, 2005, pp. 91–92. ISBN: 978-3-540-26656-3. DOI: 10.1007/11507840_11 (cit. on p. 22).

[54] *Code Aurora security advisories*. URL: https://www.codeaurora.org/projects/security-advisories — https://archive.is/SEWo8 (cit. on p. 57).

[55] Fred Cohen. "Computer viruses". PhD thesis. University of Southern California, 1985, pp. 1–152 (cit. on p. 44).

[56] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. "Breaking up is hard to do: security and functionality in a commodity hypervisor". In: *Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 2011. ISBN: 9781450309776. DOI: 10.1145/2043556.2043575 (cit. on p. 24).

[57] Mauro Conti, Vu Thien, Nga Nguyen, and Bruno Crispo. "CRePE : Context-Related Policy Enforcement for Android". In: *Lecture Notes in Computer Science (LNCS)* 6531 (2010), pp. 331–345. ISSN: 03029743. DOI: 10.1007/978-3-642-18178-8-29 (cit. on p. 47).

[58] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. "A Large-Scale Analysis of the Security of Embedded Firmwares". In: *USENIX Security Symposium* (2014) (cit. on pp. 14, 41).

[59] Jonathan Crussell, Clint Gibler, and Hao Chen. "AnDarwin: Scalable Detection of Semantically Similar Android Applications". In: *Computer Security–ESORICS*. Springer Berlin Heidelberg, 2013, pp 182–199. ISBN: 978-3-642-40202-9. DOI: 10.1007/978-3-642-40203-6_11 (cit. on p. 44).

[60] *CVE-2009-1185*. 2009. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1185 (visited on 2016-04-11) (cit. on p. 53).

[61] *CVE-2011-1149*. 2011. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1149 (visited on 2016-04-11) (cit. on p. 53).

[62] *CVE-2011-1350*. 2011. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1350 (visited on 2016-04-11) (cit. on p. 53).

[63] *CVE-2011-1352*. 2011. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1352 (visited on 2016-04-11) (cit. on p. 53).

[64] *CVE-2011-1823*. 2011. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823 (visited on 2016-04-11) (cit. on p. 53).

[65] *CVE-2011-3874*. 2011. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3874 (visited on 2016-04-11) (cit. on p. 53).

[66] *CVE-2013-4787*. 2013. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4787 (visited on 2016-04-11) (cit. on p. 53).

[67] *CVE-2014-3153*. 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153 (visited on 2016-04-11) (cit. on p. 53).

[68] *CVE-2014-7911*. 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7911 (visited on 2016-04-11) (cit. on p. 53).

[69] *CVE-2015-1538*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538 (visited on 2016-04-11) (cit. on p. 53).

[70] *CVE-2015-1539*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1539 (visited on 2016-04-11) (cit. on p. 53).

[71] *CVE-2015-3824*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3824 (visited on 2016-04-11) (cit. on p. 53).

[72] *CVE-2015-3825*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3825 (visited on 2016-04-11) (cit. on p. 53).

[73] *CVE-2015-3826*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3826 (visited on 2016-04-11) (cit. on p. 53).

[74] *CVE-2015-3827*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3827 (visited on 2016-04-11) (cit. on p. 53).

[75] *CVE-2015-3828*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3828 (visited on 2016-04-11) (cit. on p. 53).

[76] *CVE-2015-3829*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3829 (visited on 2016-04-11) (cit. on p. 53).

[77] *CVE-2015-3837*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3837 (visited on 2016-04-11) (cit. on p. 53).

[78] *CVE-2015-3876*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3876 (visited on 2016-04-11) (cit. on p. 53).

[79] *CVE-2015-6602*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6602 (visited on 2016-04-11) (cit. on p. 53).

[80] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege escalation attacks on Android". In: *Information Security Conference (ISC)* LNCS 6531 (2010), pp. 346–360. DOI: 10.1007/978-3-642-18178-8_30 (cit. on p. 41).

[81] Chris De Herrera. *Versions of Windows CE / Windows Mobile*. 2009-11. URL: http://www.pocketpcfaq.com/wce/versions.htm — https://archive.is/k0qy (visited on 2016-04-07) (cit. on p. 30).

[82] Arkajit Dey and Stephen Weis. *Keyczar: A Cryptographic Toolkit*. 2008. URL: http://keyczar.googlecode.com/files/keyczar05b.pdf (cit. on p. 29).

[83] Changyu Dong and Naranker Dulay. "Longitude: A privacy-preserving location sharing protocol for mobile applications". In: *IFIP Advances in Information and Communication Technology*. Vol. 358 AICT. Springer, 2011, pp. 133–148. ISBN: 9783642221996. DOI: 10.1007/978-3-642-22200-9_12 (cit. on p. 37).

[84] Saar Drimer. *Banks don't help fight phishing*. 2006-03. URL: https://www.lightbluetouchpaper.org/2006/03/10/banks-dont-help-fight-phishing/ — https://archive.is/tR7m7 (visited on 2015-09-08) (cit. on p. 37).

[85] Saar Drimer, Steven J. Murdoch, and Ross Anderson. "Optimised to fail: Card readers for online banking". In: *Financial Cryptography and Data Security* February (2009), pp. 184–200. DOI: 10.1007/978-3-642-03549-4_11 (cit. on p. 26).

[86] Thomas Duebendorfer and Stefan Frei. "Web browser security update effectiveness". In: *Lecture Notes in Computer Science (LNCS)* 6027 LNCS (2010), pp. 124–137. ISSN: 03029743. DOI: 10.1007/978-3-642-14379-3_11 (cit. on p. 42).

[87] Thomas Duebendorfer and Stefan Frei. *Why silent updates boost security*. Tech. rep. April. ETH Zurich, 2009 (cit. on p. 104).

[88] Thai Duong and Juliano Rizzo. *Here Come The ⊕ Ninjas*. 2011-05. URL: http://www.hpcc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf — http://blog.tempest.com.br/static/attachments/marco-carnut/driblando-ataque-beast-com-pasme-rc4/ssl_jun21.pdf (cit. on p. 28).

[89] Peter Eckersley and Jesse Burns. *Is the SSLiverse a safe place*. 2010. URL: https://ipv6.eff.org/files/ccc2010.pdf (visited on 2013-06-18) (cit. on p. 28).

[90] Stephen Elop and Steve Ballmer. *Open Letter from CEO Stephen Elop, Nokia and CEO Steve Ballmer, Microsoft*. 2011-02. URL: https://web.archive.org/web/20111231082126/http://conversations.nokia.com/2011/02/11/open-letter-from-ceo-stephen-elop-nokia-and-ceo-steve-ballmer-microsoft — https://archive.is/BUpXg (visited on 2016-04-06) (cit. on p. 30).

[91] W Enck, M Ongtang, and P McDaniel. “Understanding Android Security”. In: *IEEE Security & Privacy Magazine* 7.1 (2009), pp. 50–57. ISSN: 15407993. DOI: 10.1109/MSP.2009.26 (cit. on p. 44).

[92] William Enck. “Defending Users Against Smartphone Apps: Techniques and Future Directions”. In: *International Conference on Information Systems Security (ICISS)*. Lecture Notes in Computer Science LNCS 7093 (2011), pp. 49–70. DOI: 10.1007/978-3-642-25560-1_3 (cit. on p. 43).

[93] William Enck, Damien Octeau, Patrick Mcdaniel, and Swarat Chaudhuri. “A Study of Android Application Security”. In: *USENIX Security Symposium* August (2011), pp. 935–936. DOI: 10.1007/s00256-010-0882-8 (cit. on p. 44).

[94] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. “TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vol. 49. 4. USENIX Association, 2010, pp. 1–6 (cit. on p. 43).

[95] Ericsson. *Introduction to Erlang*. URL: http://www.erlang.org/faq/introduction.html — https://archive.is/ryeDe (visited on 2015-07-21) (cit. on p. 15).

[96] ETForecasts. *Computers-in-use forecast by country*. 2010. URL: https://archive.is/picPY — http://www.etforecasts.com/products/ES_cinusev2.htm (visited on 2015-09-24) (cit. on p. 149).

[97] Benedict Evans. *A note on install bases*. 2013-04. URL: http://benevans.com/benedictevans/2013/4/16/a-note-on-install-bases — https://archive.is/nRwlD (visited on 2015-09-24) (cit. on p. 149).

[98] Dave Evans. *The Internet of Things - How the Next Evolution of the Internet is Changing Everything*. 2011-04. URL: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (cit. on p. 13).

[99] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. “Why Eve and Mallory love Android: an analysis of Android SSL (in)security”. In: *ACM conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 50–61. ISBN: 9781450316514. DOI: 10.1145/2382196.2382205 (cit. on pp. 25, 85).

[100] Nicolas Falliere, Liam O Murchu, and Eric Chien. *W32. stuxnet dossier*. 2011. URL: http://www.h4ckr.us/library/Documents/ICS_Events/StuxnetDossier(Symantec)v1.4.pdf (visited on 2013-06-19) (cit. on p. 23).

[101] Kassem Fawaz and Kang G. Shin. "Location Privacy Protection for Smartphone Users". In: *ACM conference on Computer and Communications Security (CCS)*. ACM, 2014-11, pp. 239–250. ISBN: 9781450329576. DOI: 10.1145/2660267.2660270 (cit. on p. 37).

[102] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A survey of mobile malware in the wild". In: *ACM CCS workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2011), pp. 3–14. DOI: 10.1145/2046614.2046618 (cit. on pp. 40, 73).

[103] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, and Ariel Haney. "Android permissions: User attention, comprehension, and behavior". In: *Symposium on Usable Privacy and Security (SOUPS)*. Washington, DC, USA: ACM, 2012-07. DOI: 10.1145/2335356.2335360 (cit. on pp. 37, 49).

[104] Denzil Ferreira, Vassilis Kostakos, Alastair R. Beresford, Janne Lindqvist, and Anind K. Dey. "Securacy: An Empirical Investigation of Android Applications' Network Usage, Privacy and Security". In: *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. New York: ACM, 2015-06. ISBN: 9781450336239. DOI: 10.1145/2766498.2766506 (cit. on p. 37).

[105] Jeff Forristal. *Uncovering Android Master Key That Makes 99% of Devices Vulnerable*. 2013-03. URL: https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/ — https://archive.is/8B491 (visited on 2015-04-08) (cit. on p. 52).

[106] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. "Firefox (in)security update dynamics exposed". In: *ACM SIGCOMM Computer Communication Review* 39.1 (2008), pp. 16–22 (cit. on p. 40).

[107] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. "Large-scale vulnerability analysis". In: *SIGCOMM workshop on Large-scale attack defense (LSAD)*. ACM, 2006-09, pp. 131–138. ISBN: 1595934170 (cit. on p. 40).

[108] Stefan Frei, Dominik Schatzmann, Bernhard Plattner, and Brian Trammell. "Modeling the Security Ecosystem – The Dynamics of (In)Security". In: *Economics of Information Security and Privacy*. Springer, 2010. Chap. 6, pp. 79–106. ISBN: 978-1-4419-6967-5. DOI: 10.1007/978-1-4419-6967-5_6 (cit. on pp. 54, 55).

[109] FTC. *FTC Approves Final Order Settling Charges Against HTC America Inc.* 2013. URL: https://www.ftc.gov/news-events/press-releases/2013/07/ftc-approves-final-order-settling-charges-against-htc-america-inc — https://www.ftc.gov/sites/default/files/documents/cases/2013/07/130702htcdo.pdf (cit. on p. 113).

[110] Huiqing Fu and Janne Lindqvist. "General Area or Approximate Location? How People Understand Location Permissions". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM, 2014-11, pp. 117–120. ISBN: 9781450331487. DOI: 10.1145/2665943.2665957 (cit. on p. 37).

[111] Huiqing Fu, Yulong Yang, Nileema Shingte, Janne Lindqvist, and Marco Gruteser. "A Field Study of Run-Time Location Access Disclosures on Android Smartphones". In: *USEC* February (2014). DOI: 10.14722/usec.2014.23044 (cit. on p. 37).

[112] Gerhard De Koning Gans, Jaap-henk Hoepman, and Flavio D Garcia. "A practical Attack on the MIFARE Classic". In: (2008), pp. 267–282 (cit. on p. 14).

[113] David Garlan, Robert Allen, and John Ockerbloom. "Architectural Mismatch: Why Reuse Is Still So Hard". In: *IEEE Software* 26.4 (2009), pp. 66–69. ISSN: 07407459. DOI: 10.1109/MS.2009.86 (cit. on p. 29).

[114] Gartner. *Gartner Says a Typical Family Home Could Contain More Than 500 Smart Devices by 2022*. 2014-09. URL: http://www.gartner.com/newsroom/id/2839717 — https://archive.is/IniBX (visited on 2015-07-21) (cit. on p. 13).

[115] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks". In: *Network and Distributed System Security Symposium (NDSS)* (2014). DOI: 10.14722/ndss.2014.23323 (cit. on pp. 42, 88).

[116] Christos Gkantsidis, Thomas Karagiannis, and Milan Vojnović. "Planet scale software updates". In: *ACM SIGCOMM Computer Communication Review* 36 (2006), pp. 423–434. ISSN: 01464833. DOI: 10.1145/1151659.1159961 (cit. on pp. 42, 74).

[117] Daniel Gold. *Unit 61398: Chinese Cyber-Espionage and the Advanced Persistent Threat*. 2013-03. URL: http://resources.infosecinstitute.com/unit-61398-chinese-cyber-espionage-and-the-advanced-persistent-threat/ — https://archive.is/EetIk (visited on 2015-09-08) (cit. on p. 23).

[118] Ian Goldberg and David Wagner. "Randomness and the Netscape browser". In: *Dr Dobb's Journal* (1996) (cit. on p. 28).

[119] Li Gong and Roger M. Needham. "Reasoning about belief in cryptographic protocols". In: *IEEE Symposium on Research in Security and Privacy*. Oakland, California: IEEE, 1990, pp. 234–248. DOI: 10.1109/RISP.1990.63854 (cit. on p. 29).

[120] Dan Goodin. *New Linux worm targets routers, cameras, "Internet of things" devices*. 2013-11. URL: http://arstechnica.com/security/2013/11/new-linux-worm-targets-routers-cameras-Internet-of-things-devices/ — https://archive.is/eJlLe (visited on 2015-07-20) (cit. on p. 14).

[121] Google. *Codelines, Branches, and Releases*. URL: https://source.android.com/source/code-lines.html — https://archive.is/0kqcf (visited on 2015-09-17) (cit. on p. 95).

[122] Google. *Get Started with Publishing*. 2015. URL: http://developer.android.com/distribute/googleplay/start.html — https://archive.is/Xn7Zf (visited on 2015-07-21) (cit. on p. 115).

[123] Google. *Jelly Bean version information*. 2015. URL: https://developer.android.com/about/versions/jelly-bean.html — https://archive.is/pHvjl (visited on 2015-04-08) (cit. on p. 115).

[124] Google and Adrian Ludwig. *Android Security 2014 Year in Review*. 2015-04. URL: https://source.android.com/devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf — http://googleonlinesecurity.blogspot.com/2015/04/android-security-state-of-union-2014.html (cit. on pp. 17, 43, 45, 51, 60, 75, 115).

[125] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection". In: *International conference on mobile systems, applications, and services (MobiSys)*. 2012, pp. 281–293. ISBN: 9781450313018. DOI: 10.1145/2307636.2307663 (cit. on p. 44).

[126] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. “Systematic detection of capability leaks in stock Android smartphones”. In: *Network and Distributed System Security Symposium (NDSS)* (2012) (cit. on pp. 41, 45, 92).

[127] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. “Unsafe exposure analysis of mobile in-app advertisements”. In: *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)* (2012), pp. 101–112. DOI: 10.1145/2185448.2185464 (cit. on p. 88).

[128] Andy Greenberg. *Hackers remotely kill a jeep on the highway—with me in it*. 2015-07. URL: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ — https://archive.is/pEpnP (visited on 2015-07-21) (cit. on p. 14).

[129] Glenn Greenwald and Ewen MacAskill. *NSA Prism program taps in to user data of Apple, Google and others*. 2013-06. URL: http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data — https://archive.is/VBu6X (visited on 2015-09-08) (cit. on p. 23).

[130] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Huijie Deng. “Comparing mobile privacy protection through cross-platform applications”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA, 2013 (cit. on p. 39).

[131] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. “Mining your Ps and Qs: detection of widespread weak keys in network devices”. In: *Proceedings of the 21st conference on Security symposium*. USENIX, 2012 (cit. on p. 28).

[132] Historylearningsite. *The Personal Computer*. URL: http://www.historylearningsite.co.uk/inventions-and-discoveries-of-the-twentieth-century/the-personal-computer/ — https://archive.is/Tlnqg (visited on 2015-09-24) (cit. on p. 149).

[133] Sipho Hlongwane. *BBX, BlackBerry’s latest purported saviour*. 2011-10. URL: http://www.dailymaverick.co.za/article/2011-10-21-bbx-blackberrys-latest-purported-saviour — https://archive.is/FgjTn (visited on 2016-04-07) (cit. on p. 30).

[134] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. “PREC: Practical Root Exploit Containment for Android Devices”. In: *ACM Conference on Data and Application Security and Privacy* (2014), pp. 187–198. DOI: 10.1145/2557547.2557563 (cit. on p. 47).

[135] Chris Hoffman. *HTG Explains: Why Android Geeks Buy Nexus Devices*. 2013-05. URL: http://www.howtogeek.com/139391/htg-explains-why-android-geeks-buy-nexus-devices/ — https://archive.is/iGxrZ (visited on 2015-09-17) (cit. on p. 99).

[136] Russell Holly. *Keeping apps from seeing your data*. 2015-05. URL: http://www.androidcentral.com/cyanogen-os-privacy-guard-keeping-apps-seeing-your-data — https://archive.is/XmVv0 (visited on 2016-03-03) (cit. on p. 38).

[137] HTC. *The anatomy of an Android OS update*. 2013. URL: http://www.htc.com/us/go/htc-software-updates-process/ — https://archive.is/I92W4 (visited on 2015-06-03) (cit. on p. 92).

[138] Google Inc. *Google Play developer dashboard*. 2016-03. URL: https://developer.android.com/about/dashboards/index.html (visited on 2016-03-23) (cit. on p. 78).

[139] Internetlivestats. *Internet users*. 2014. URL: http://www.internetlivestats.com/internet-users/ — https://archive.is/9MLtc (cit. on p. 149).

[140] William Jackson. *The hard part of DNS security lies beyond the next deadline*. 2009-07. URL: https://gcn.com/articles/2009/07/13/dnssec-6-month-deadline-key-management.aspx (visited on 2016-03-03) (cit. on p. 28).

[141] Xing Jin, Tongbo Luo, Derek G Tsui, and Wenliang Du. "Code Injection Attacks on HTML5-based Mobile Apps". In: *Workshop on Mobile Security Technologies (MoST)* (2014). DOI: 10.1145/2660267.2660275 (cit. on p. 42).

[142] Karen Sparck Jones. *A brief informal history of the Computer Laboratory*. 1999. URL: http://www.cl.cam.ac.uk/events/EDSAC99/history.html — https://archive.is/zlwJr (visited on 2015-09-28) (cit. on p. 151).

[143] Wolfgang Kandek. *Patch Tuesday May 2015*. 2015-05. URL: https://community.qualys.com/blogs/laws-of-vulnerabilities/2015/05/12/patch-tuesday-may-2015 — https://archive.is/78Lg5 (visited on 2015-08-19) (cit. on p. 14).

[144] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. "A conundrum of permissions: Installing applications on an Android smartphone". In: *Financial Cryptography Workshops, USEC and WECSR* 7398 LNCS (2012), pp. 68–79. ISSN: 03029743. DOI: 10.1007/978-3-642-34638-5_6 (cit. on p. 37).

[145] Martin Kleppmann and Conrad Irwin. “Octokey: Public key authentication for the web”. 2015 (cit. on p. 24).

[146] Karl Koscher et al. “Experimental security analysis of a modern automobile”. In: *IEEE Symposium on Security and Privacy* (2010), pp. 447–462. ISSN: 10816011. DOI: 10.1109/SP.2010.34 (cit. on p. 14).

[147] Daniel E. Krych, Stephen Lange-Maney, Patrick McDaniel, and William Glodek. “Investigating weaknesses in Android certificate security”. In: *Modeling and Simulation for Defense Systems and Applications* SPIE 9478 (2015), 947804:1–947804:9. DOI: 10.1117/12.2177498 (cit. on p. 39).

[148] Oxford University Computing Laboratory. *Computing Laboratory*. 1996. URL: http://www.oua.ox.ac.uk/holdings/ComputingLaboratory2CL.pdf (cit. on p. 151).

[149] MWR Labs. *WebView addJavascriptInterface Remote Code Execution*. 2013. URL: https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/ — https://archive.is/KxtXb (visited on 2014-12-19) (cit. on p. 88).

[150] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. “L4Android: A Generic Operating System Framework for Secure Smartphones”. In: *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2011), pp. 39–50. ISSN: 15437221. DOI: 10.1145/2046614.2046623 (cit. on p. 47).

[151] Ralph Langner. “Stuxnet: Dissecting a cyberwarfare weapon”. In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51. DOI: 10.1109/MSP.2011.67 (cit. on p. 23).

[152] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. “Finding error handling bugs in OpenSSL using Coccinelle”. In: *Dependable Computing Conference (EDCC)*. 2010, pp. 191–196. DOI: 10.1109/EDCC.2010.31 (cit. on pp. 26, 41).

[153] Mike Lennon. *Google Lets SMTP Certificate Expire*. 2015-04. URL: http://www.securityweek.com/google-lets-smtp-certificate-expire — https://archive.is/vktDw (visited on 2016-03-03) (cit. on p. 28).

[154] Lawrence Lessig. *CODE version 2.0*. 2nd ed. Basic Books, 2006, pp. 1–424. ISBN: 9780465039142 (cit. on p. 16).

[155] Bo Li, Elena Reshetova, and Tuomas Aura. “Symbian OS platform security model”. In: *;LOGIN:* (2010) (cit. on p. 38).

[156] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. “ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors”. In: *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. Wroclaw, Poland, 2014-09 (cit. on p. 44).

[157] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. “Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps”. In: *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. Florence, Italy: ACM, 2015-05, pp. 89–103. ISBN: 9781450334945. DOI: 10.1145/2742647.2742668 (cit. on p. 47).

[158] Ben Lovejoy. *Report: Apple takes 92% of smartphone market profits on just 20% of sales*. 2015-07. URL: http://9to5mac.com/2015/07/13/apple-smartphone-profit-share/ — https://archive.is/OAQ0I (visited on 2016-04-08) (cit. on p. 33).

[159] Adrian Ludwig and Venkat Rapaka. *An Update to Nexus Devices*. 2015-08. URL: http://officialandroid.blogspot.jp/2015/08/an-update-to-nexus-devices.html — https://archive.is/yfa7o (visited on 2015-09-28) (cit. on p. 93).

[160] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. “Attacks on WebView in the Android System”. In: *Annual Computer Security Applications Conference (ACSAC)*. Orlando, Florida, USA: ACM, 2011, pp. 343–352. ISBN: 9781450306720. DOI: 10.1145/2076732.2076781 (cit. on p. 88).

[161] John Lyle, Shamal Faily, Ivan Fléchais, André Paul, Ayşe Göker, Hans Myrhaug, Heiko Desruelle, and Andrew Martin. “On the design and development of webinos: A distributed mobile application middleware”. In: *Distributed Applications and Interoperable Systems (DAIS)* 7272 LNCS (2012), pp. 140–147. ISSN: 03029743. DOI: 10.1007/978-3-642-30823-9_12 (cit. on p. 16).

[162] Anil Madhavapeddy, Richard Mortier, and Charalampos Rotsos. “Unikernels: Library Operating Systems for the Cloud”. In: *International conference on Architectural support for programming languages and operating systems (ASPLOS)*. Houston, Texas, 2013. ISBN: 9781450318709 (cit. on pp. 17, 22).

[163] Anil Madhavapeddy et al. “Jitsu: Just-in-time summoning of unikernels”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, CA, USA: USENIX, 2015-05, pp. 559–573. ISBN: 9781931971218 (cit. on p. 15).

[164] Jim Mallery. *Your home electrical system: how long can it last?* 2013-04. URL: http://www.improvementcenter.com/electrical/home-electrical-system-how-long-can-it-last.html — https://archive.is/DAORD (visited on 2016-03-03) (cit. on p. 14).

[165] Fabio Massacci and Viet Hung Nguyen. “Which is the right source for vulnerability studies? An empirical analysis on Mozilla Firefox”. In: *International Workshop on Security Measurements and Metrics (MetriSec)*. Bolzano-Bozen, Italy: ACM, 2010-09. ISBN: 9781450303408 (cit. on p. 40).

[166] Neil Mawston. *Android Shipped 1 Billion Smartphones Worldwide in 2014*. 2015-01. URL: https://www.strategyanalytics.com/strategy-analytics/blogs/devices/smartphones/smartphones/2015/03/11/android-shipped-1-billion-smartphones-worldwide-in-2014 — https://archive.is/eHb74 (visited on 2015-09-28) (cit. on pp. 30, 152).

[167] Miles A. McQueen, Trevor A. McQueen, Wayne F. Boyer, and May R. Chaffin. “Empirical estimates and observations of 0Day vulnerabilities”. In: *Annual Hawaii International Conference on System Sciences (HICSS)* (2009), pp. 1–12. ISSN: 1530-1605. DOI: 10.1109/HICSS.2009.186 (cit. on p. 40).

[168] Mary Meeker. *Internet trends 2015 – code conference*. 2015-05. URL: http://www.kpcb.com/file/kpcb-internet-trends-2015 (visited on 2015-09-28) (cit. on p. 152).

[169] Adrian Mettler, Daniel Wagner, and Tyler Close. “Joe-E: A security-oriented subset of Java”. In: *Network and Distributed System Security Symposium (NDSS)*. 2010 (cit. on p. 88).

[170] Charlie Miller and Chris Valasek. *Adventures in Automotive Networks and Control Units*. 2013. URL: http://can-newsletter.org/assets/files/ttmedia/raw/c51e81bf7c09578c37e3f7a1f97c197b.pdf — http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf (cit. on p. 14).

[171] Dan Morrill. *Announcing the Android 1.0 SDK, release 1*. 2008-09. URL: http://android-developers.blogspot.co.uk/2008/09/announcing-android-10-sdk-release-1.html — https://archive.is/z8jfc (visited on 2016-04-07) (cit. on p. 30).

[172] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. "PatchDroid: scalable third-party security patches for Android devices". In: *Annual Computer Security Applications Conference (ACSAC)* (2013), pp. 259–268. DOI: 10.1145/2523649.2523679 (cit. on p. 42).

[173] Stephen J. Murdoch. "Hardened Stateless Session Cookies". In: *Security Protocols XVI* 6615.April (2011), pp. 93–101. DOI: 10.1007/978-3-642-22137-8_13 (cit. on p. 26).

[174] Steven J. Murdoch and Ross Anderson. "Verified by Visa and MasterCard SecureCode: or, how not to design authentication". In: *Financial Cryptography and Data Security* January (2010), pp. 336–342. DOI: 10.1007/978-3-642-14577-3_27 (cit. on p. 26).

[175] Yacin Nadji, Jonathon Giffin, and Patrick Traynor. "Automated remote repair for mobile malware". In: *Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 413–422. DOI: 10.1145/2076732.2076791 (cit. on p. 44).

[176] Adwait Nadkarni, Vasant Tendulkar, and William Enck. "NativeWrap: Ad Hoc Smartphone Application Creation for End Users". In: *ACM conference on Security and privacy in wireless & mobile networks (WiSec)*. ACM, 2014, pp. 13–24. ISBN: 9781450329729. DOI: 10.1145/2627393.2627412 (cit. on p. 42).

[177] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. "The Attack of the Clones : A Study of the Impact of Shared Code on Vulnerability Patching". In: *IEEE Symposium on Security and Privacy* (2015), pp. 692–708. DOI: 10.1109/SP.2015.48 (cit. on p. 40).

[178] Dave Neal. *Kaspersky: The Equation Group looks and smells even more like the NSA*. 2015-03. URL: http://www.theinquirer.net/inquirer/news/2395638/kaspersky-fingers-nsa-style-equation-group-for-hard-drive-backdoor-epidemic — https://archive.is/n9lFW (cit. on p. 23).

[179] Netmarketshare. *Desktop operating system market share*. 2014-07. URL: https://archive.is/PLGxn (visited on 2016-04-07) (cit. on p. 74).

[180] Netmarketshare. *Desktop operating system market share*. 2015-06. URL: https://archive.is/xQkls (visited on 2016-04-07) (cit. on p. 74).

[181] Yossef Oren and Angelos D Keromytis. "From the Aether to the Ethernet – Attacking the Internet using Broadcast Digital Television". In: *USENIX Security Symposium*. May. San Diego, CA, USA: USENIX, 2014-08. ISBN: 9781931971157 (cit. on p. 14).

[182] Eric Osterweil, Burt Kaliski, Matt Larson, and Danny Mcpherson. "Reducing the X.509 Attack Surface with DNSSEC's DANE". In: *Securing and Trusting Interent Names (SATIN)*. NPL, 2012-03 (cit. on p. 24).

[183] Andy Ozment. "Bug auctions: Vulnerability markets reconsidered". In: *Workshop on Economics of Information Security (WEIS)*. Minneapolis, MN, USA, 2004-05, pp. 1–23 (cit. on p. 41).

[184] Andy Ozment and Stuart E Schechter. "Milk or Wine: Does Software Security Improve with Age?" In: *USENIX Security Symposium* (2006), pp. 93–104 (cit. on p. 41).

[185] Steven Max Patterson. *Contrary to what you've heard, Android is almost impenetrable to malware*. 2013-10. URL: http://qz.com/131436/contrary-to-what-youve-heard-android-is-almost-impenetrable-to-malware/ — https://archive.is/FrNFi (visited on 2015-03-26) (cit. on p. 60).

[186] Lawrence C. Paulson. "Inductive analysis of the Internet protocol TLS". In: *ACM Transactions on Information and System Security* 2.3 (1999-08), pp. 332–351. ISSN: 10949224. DOI: 10.1145/322510.322530 (cit. on p. 29).

[187] Paul Pearce, Adrienne Porter Felt, and David Wagner. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *ACM Symposium on Information, Computer and Communication Security (ASIACCS)* (2012). DOI: 10.1145/2414456.2414498 (cit. on p. 88).

[188] John Pescatore. *Nimda Worm Shows You Can't Always Patch Fast Enough*. 2001. URL: https://www.gartner.com/doc/340962 — https://archive.today/tdhfu (cit. on p. 40).

[189] Christy Pettey. *Gartner Says Hewlett-Packard Takes Clear Lead in Fourth Quarter Worldwide PC Shipments and Creates a Virtual Tie with Dell for 2006 Year-End Results*. 2007-01. URL: http://www.gartner.com/newsroom/id/500384 — https://archive.is/X4ouI (visited on 2015-09-28) (cit. on p. 151).

[190] Christy Pettey. *Gartner Says Worldwide PC Market Grew 13 Percent in 2007*. 2008-01. URL: http://www.gartner.com/newsroom/id/584210 — https://archive.is/kSsWl (visited on 2015-09-28) (cit. on p. 151).

[191] Christy Pettey. *Gartner Says Worldwide PC Shipments in Fourth Quarter of 2010 Grew 3.1 Percent; Year-End Shipments Increased 13.8 Percent*. 2011-01. URL: http://www.gartner.com/newsroom/id/1519417 — https://archive.is/BAOAx (visited on 2015-09-28) (cit. on p. 151).

[192] Christy Pettey and Laurence Goasduff. *Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010*. 2011-02. URL: http://www.gartner.com/newsroom/id/1543014 — https://archive.is/S5okJ (cit. on p. 152).

[193] Christy Pettey and Laurence Goasduff. *Gartner Says Worldwide Mobile Phone Sales to End Users Grew 8 Per Cent in Fourth Quarter 2009; Market Remained Flat in 2009*. 2010-02. URL: http://www.gartner.com/newsroom/id/1306513 — https://archive.is/YPGqY (visited on 2015-09-28) (cit. on p. 152).

[194] Christy Pettey and Rob van der Meulen. *Gartner Says In the Fourth Quarter of 2008 the PC Industry Suffered Its Worst Shipment Growth Rate Since 2002*. 2009-01. URL: http://www.gartner.com/newsroom/id/856712 — https://archive.is/S0BoJ (visited on 2015-09-28) (cit. on p. 151).

[195] Christy Pettey and Rob van der Meulen. *Gartner Says Worldwide PC Shipments in Fourth Quarter of 2009 Posted Strongest Growth Rate in Seven Years*. 2010-01. URL: http://www.gartner.com/newsroom/id/1279215 — https://archive.is/Lc6Da (visited on 2015-09-28) (cit. on p. 151).

[196] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications". In: *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014-11, pp. 1354–1365. ISBN: 9781450329576. DOI: 10.1145/2660267.2660287 (cit. on p. 37).

[197] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. "DroidChameleon: evaluating Android Anti-malware against Transformation Attacks". In: *Symposium on Information, Computer and Communications Security (ASIA CCS)*. March. ACM, 2013, pp. 329–334. ISBN: 978-1-4503-1767-2. DOI: 10.1145/2484313.2484355 (cit. on p. 44).

[198] Eric Ravenscraft. *Why Google Play Services are now more important than Android*. 2013-07. URL: http://lifehacker.com/why-google-play-services-are-now-more-important-than-an-975970197 — https://archive.is/IFRJP (visited on 2015-07-22) (cit. on p. 115).

[199] Eric S. Raymond. *The Cathedral and the Bazaar*. 1st. O'Reilly & Associates, Inc., 1999. ISBN: 1565927249 (cit. on p. 114).

[200] Joel Reardon, Srdjan Capkun, and David Basin. "Data node encrypted file system: Efficient secure deletion for flash memory". In: *USENIX Security Symposium*. 2012 (cit. on p. 48).

[201] Eric Rescorla. "Is finding security holes a good idea?" In: *IEEE Security and Privacy* 3.1 (2005), pp. 14–19. ISSN: 15407993. DOI: 10.1109/MSP.2005.17 (cit. on p. 41).

[202] Thomas Ricker. *Microsoft announces ten Windows Phone 7 handsets for 30 countries: October 21 in Europe and Asia, 8 November in US (Update: Video!)* 2010-10. URL: http://www.engadget.com/2010/10/11/microsoft-announces-ten-windows-phone-7-handsets-for-30-countrie/ — https://archive.is/axiM (visited on 2016-04-07) (cit. on p. 30).

[203] Salvador Rodriguez. *Refrigerator among devices hacked in Internet of Things cyber attack*. 2014-01. URL: http://articles.latimes.com/2014/jan/16/business/la-fi-tn-refrigerator-hacked-internet-of-things-cyber-attack-20140116 — https://archive.is/bVRgB (visited on 2015-07-20) (cit. on p. 14).

[204] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems". In: *2012 IEEE Symposium on Security and Privacy* (2012-05), pp. 224–238. DOI: 10.1109/SP.2012.24 (cit. on p. 38).

[205] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. ISSN: 00189219. DOI: 10.1109/PROC.1975.9939 (cit. on p. 24).

[206] *Samsung Announces an Android Security Update Process to Ensure Timely Protection from Security Vulnerabilities*. 2015-08. URL: http://global.samsungtomorrow.com/samsung-announces-an-android-security-update-process-to-ensure-timely-protection-from-

security-vulnerabilities/ — https://archive.is/m1e02 (visited on 2015-09-28) (cit. on p. 93).

[207] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Leonid Batyuk, Jan Hendrik Clausen, Seyit Ahmet Camtepe, Sahin Albayrak, and Can Yildizli. "Smartphone malware evolution revisited: Android next target?" In: *International Conference on Malicious and Unwanted Software (MALWARE)* (2009), pp. 1–7. ISSN: 10557903. DOI: 10.1109/MALWARE.2009.5403026 (cit. on p. 43).

[208] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yuksel, Seyit A. Camtepe, and Sahin Albayrak. "Static Analysis of Executables for Collaborative Malware Detection on Android". In: *IEEE International Conference on Communications*. IEEE, 2009-06, pp. 1–5. DOI: 10.1109/ICC.2009.5199486 (cit. on p. 43).

[209] Bruce Schneier. *Feudal security*. 2012-12. URL: http://www.schneier.com/blog/archives/2012/12/feudal_sec.html — https://archive.is/QdDrg (cit. on pp. 15, 23).

[210] Bruce Schneier, Matthew Fredrikson, Tadayoshi Kohno, and Thomas Ristenpart. *Surreptitiously weakening cryptographic systems*. Tech. rep. 2015-02, pp. 1–26 (cit. on p. 25).

[211] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating smartphone advertising from applications". In: *USENIX Security symposium* (2012), pp. 1–15. arXiv: 1202.4030 (cit. on p. 88).

[212] Steve Sheng, Levi Broderick, Jeremy J. Hyland, and Colleen Alison Koranda. "Why Johnny still can't encrypt: evaluating the usability of email encryption software". In: *Symposium On Usable Privacy and Security (SOUPS)*. 2006, pp. 3–4 (cit. on p. 36).

[213] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android". In: *Network and Distributed System Security Symposium (NDSS)* (2013) (cit. on pp. 46, 115).

[214] *Smartphone worldwide installed base by operating system*. URL: http://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/ — https://archive.is/cyRMn (cit. on p. 15).

[215] Christopher Soghoian and Ben Wizner. *ACLU FTC Android updates*. 2013. URL: http://www.aclu.org/files/assets/aclu_-_android_ftc_complaint_-_final.pdf (cit. on pp. 42, 93, 113).

[216] WebdesignerDepot Staff. *The Evolution of Cell Phone Design Between 1983-2009*. 2009-05. URL: http://www.webdesignerdepot.com/2009/05/the-evolution-of-cell-phone-design-between-1983-2009/ — https://archive.is/evneQ (visited on 2016-04-06) (cit. on p. 30).

[217] Frank Stajano. "Pico: No more passwords!" In: *Security Protocols XIX* LNCS 7114 (2011), pp. 49–81. DOI: 10.1007/978-3-642-25867-1_6 (cit. on p. 14).

[218] Frank Stajano and Ross Anderson. "The Resurrecting Duckling : Security Issues for Ad-hoc Wireless Networks". In: *Security Protocols*. LNCS 1796 (1999), pp. 172–182. DOI: 10.1007/10720107_24 (cit. on p. 15).

[219] Emily Stark, Michael Hamburg, and Dan Boneh. "Symmetric Cryptography in Javascript". In: *Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2009-12, pp. 373–381. ISBN: 978-1-4244-5327-6. DOI: 10.1109/ACSAC.2009.42 (cit. on p. 28).

[220] Statsia. *Global installed base of desktop PC from 2005 to 2015 (in 1,000 units)*. 2011-03. URL: http://www.statista.com/statistics/203585/global-installed-base-of-desktop-pcs/ — https://archive.is/cHIAu (visited on 2015-09-28) (cit. on p. 151).

[221] Statsia. *Global installed base of mobile PC from 2005 to 2015 (in 1,000 units)*. 2011-03. URL: http://www.statista.com/statistics/203617/global-installed-base-of-mobile-pcs/ — https://archive.is/M4oa8 (visited on 2015-09-28) (cit. on p. 151).

[222] Statsia. *Global smartphone sales to end users from 1st quarter 2009 to 3rd quarter 2014, by operating system (in million units)*. 2014. URL: http://www.statista.com/statistics/266219/global-smartphone-sales-since-1st-quarter-2009-by-operating-system/ — https://archive.is/dH6w9 (visited on 2015-09-28) (cit. on pp. 30, 152).

[223] Statsia. *Smartphones worldwide installed base from 2008 to 2017 (in millions)*. 2014-11. URL: http://www.statista.com/statistics/371889/smartphone-worldwide-

installed-base/ — https://archive.is/TM17p (visited on 2015-09-28) (cit. on p. 152).

[224] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. "Investigating user privacy in Android ad libraries". In: *IEEE Mobile Security Technologies (MoST)*. 2012 (cit. on p. 88).

[225] Tim Stöber, Mario Frank, Jens Schmitt, and Ivan Martinovic. "Who do you sync you are? Smartphone fingerprinting via application behaviour". In: *Security and privacy in wireless and mobile networks (WiSec)* (2013), pp. 7–12. DOI: 10.1145/2462096.2462099 (cit. on p. 17).

[226] *Symbian*. 2013. URL: http://www.gsmarena.com/glossary.php3?term=symbian — https://archive.is/bZbPE (visited on 2016-04-06) (cit. on p. 30).

[227] Paul F. Syverson and Paul C. van Oorschot. "On unifying some cryptographic protocol logics". In: *IEEE Computer Society Symposium on Research in Security and Privacy*. 1994, pp. 14–28. DOI: 10.1109/RISP.1994.296595 (cit. on p. 29).

[228] Yang Tang, Phillip Ames, Sravan Bhamidipati, and Ashish Bijlani. "CleanOS: Limiting mobile data exposure with idle eviction". In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX, 2012 (cit. on p. 48).

[229] John R. Taylor. *An introduction to error analysis*. 2nd ed. University Science Books Sausalito, California, 1997. ISBN: 093570275X (cit. on p. 98).

[230] Daniel R. Thomas. *Android release dates*. 2015-12. URL: http://androidvulnerabilities.org/release_dates.json — https://archive.is/itWLd (visited on 2016-04-09) (cit. on p. 56).

[231] Daniel R. Thomas. *Historic Google Play Dashboard*. 2015. URL: http://androidvulnerabilities.org/play/historicplaydashboard — https://archive.is/9Yfyf (visited on 2015-09-26) (cit. on pp. 61, 69, 78).

[232] Daniel R. Thomas and Alastair R. Beresford. *AndroidVulnerabilities.org*. 2015. URL: http://androidvulnerabilities.org/ — https://archive.is/VdiPu (cit. on pp. 19, 49, 54, 95).

[233] Daniel R. Thomas and Alastair R. Beresford. “Better authentication: Password revolution by evolution”. In: *Security Protocols XXII*. Vol. 8809. Cambridge, UK: Springer, 2014-03, pp. 130–145. ISBN: 978-3-319-12399-8. DOI: 10.1007/978-3-319-12400-1_13 (cit. on pp. 18, 24).

[234] Daniel R. Thomas and Alastair R. Beresford. “Incentivising software updates”. In: *Internet of Things Software Updates Workshop (IoTSU)*. Dublin, Ireland, 2016-06 (cit. on pp. 19, 93).

[235] Daniel R. Thomas and Alastair R. Beresford. *Nigori: Secrets in the cloud*. 2013. URL: http://www.cl.cam.ac.uk/research/dtg/nigori/ (visited on 2013-01-01) (cit. on p. 28).

[236] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. “Security metrics for the Android ecosystem”. In: *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. Denver, Colorado, USA: ACM, 2015-10. ISBN: 978-1-4503-3819-6. DOI: 10.1145/2808117.2808118 (cit. on pp. 19, 50, 60, 93).

[237] Daniel R. Thomas, Thomas Coudray, and Tom Sutcliffe. *Supporting data for: “The lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface”*. 2015-05. URL: https://www.repository.cam.ac.uk/handle/1810/247976 (visited on 2015-05-26) (cit. on p. 19).

[238] Daniel R. Thomas, Daniel T. Wagner, Alastair R. Beresford, and Andrew Rice. *Supporting data for: “Security metrics for the Android ecosystem”*. 2015-07. URL: https://www.repository.cam.ac.uk/handle/1810/249077 (visited on 2015-07-27) (cit. on p. 19).

[239] Daniel R. Thomas, Alastair R. Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. “The lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface”. In: *Security Protocols XXIII*. Springer, 2015-03 (cit. on pp. 18, 78).

[240] Ken Thompson. “Reflections on trusting trust”. In: *Communications of the ACM* 27.8 (1984-08), pp. 761–763. ISSN: 00010782. DOI: 10.1145/358198.358210 (cit. on p. 24).

[241] Kami E. Vaniea, Emilee Rader, and Rick Wash. “Betrayed by updates: How negative experiences affect future security”. In: *ACM conference on Human factors in computing systems (CHI)* (2014), pp. 2671–2674. DOI: 10.1145/2556288.2557275 (cit. on p. 112).

[242] Timothy Vidas and Nicolas Christin. “Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces”. In: *Data and Application Security and Privacy CODASPY*. San Antonio, Texas, USA: ACM, 2013-02, pp. 197–207. ISBN: 9781450318907. DOI: 10.1145/2435349.2435378 (cit. on p. 39).

[243] Timothy Vidas, Daniel Votipka, and Nicolas Christin. “All Your Droid Are Belong To Us: A Survey of Current Android Attacks”. In: *USENIX conference on Offensive technologies (WOOT)* (2011), pp. 1–10 (cit. on p. 40).

[244] Nicolas Viennot, Edward Garcia, and Jason Nieh. “A measurement study of Google Play”. In: *SIGMETRICS* (2014). DOI: 10.1145/2591971.2592003 (cit. on pp. 44, 89).

[245] Daniel T. Wagner. *Device Analyzer collected information*. 2011. URL: https://deviceanalyzer.cl.cam.ac.uk/collected.htm — https://archive.is/hviaV (visited on 2016-03-02) (cit. on pp. 35, 61).

[246] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. “Device Analyzer: Large-scale mobile data collection”. In: *Sigmetrics, Big Data Workshop*. Pittsburgh, PA: ACM, 2013-06. DOI: 10.1145/2627534.2627553 (cit. on pp. 61, 85, 93, 95).

[247] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. “Device Analyzer: Understanding smartphone usage”. In: *International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MOBIQUITOUS)* (2013), pp. 1–12. ISSN: 1867-8211. DOI: 10.1007/978-3-319-11569-6_16 (cit. on pp. 19, 35).

[248] David Wagner and Dean Tribble. *A Security Analysis of the Combex DarpaBrowser Architecture*. Tech. rep. 2002 (cit. on p. 88).

[249] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. “Jekyll on iOS: when benign apps become evil”. In: *USENIX Security Symposium* (2013), pp. 559–572 (cit. on p. 51).

[250] Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. “Out of the Loop: How Automated Software Updates Cause Unintended Security Consequences”. In: *Symposium on Usable Privacy and Security (SOUPS)* (2014) (cit. on p. 112).

[251] Robert N. M. Watson, Jonathan Anderson, Kris Kennaway, and Ben Laurie. "Capsicum: practical capabilities for UNIX". In: *USENIX Security Symposium*. Vol. 46. 2. USENIX Association, 2010-08, pp. 29–46 (cit. on pp. 17, 47).

[252] Fengguo Wei, Sankardas Roy, and Xinming Ou. "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps". In: *Computer and Communications Security (CCS)*. ACM, 2014-11, pp. 1329–1341. ISBN: 9781450329576 (cit. on p. 44).

[253] *Welcome to CyanogenMod*. URL: `http://www.cyanogenmod.org/` (visited on 2016-03-03) (cit. on p. 38).

[254] Alma Whitten and J.D. Tygar. "Why Johnny Can't Encrypt". In: *USENIX Security Symposium*. Washington, D.C., 1999-08, pp. 679–702 (cit. on p. 36).

[255] Wikipedians. *World Population*. 2015. URL: `https://en.wikipedia.org/wiki/World_population` — `https://archive.is/05sJl` (cit. on p. 149).

[256] Maurice V. Wilkes. *Time-sharing computer systems*. 2nd ed. MacDonald & Co., 1968. ISBN: 0356024261 (cit. on p. 15).

[257] Maurice V. Wilkes, David J. Wheeler, and Stanley Gill. *Preparation of programs for an electronic digital computer: with special reference to the EDSAC and the use of a library of subroutines*. Cambridge, Mass: Addison Wesley, 1951, pp. 1–167 (cit. on p. 13).

[258] Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen. "Static Analysis of Dalvik Bytecode and Reflection in Android". In: *Master's thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark* (2012) (cit. on p. 88).

[259] *Worldwide software developer numbers*. URL: `http://www.fiercedeveloper.com/story/evans-data-mobile-developers-now-number-87-million-worldwide/2014-06-20` — `https://archive.is/mK9hP` (visited on 2015-09-26) (cit. on p. 13).

[260] Jason L. Wright. "Software vulnerabilities: lifespans, metrics, and case study". PhD thesis. University of Idaho, 2014 (cit. on pp. 97, 98).

[261] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. "AirBag: Boosting Smartphone Resistance to Malware Infection". In: *NDSS* February (2014), pp. 23–26. DOI: `10.14722/ndss.2014.23164` (cit. on p. 47).

[262] Zhen Xie and Sencun Zhu. “AppWatcher: Unveiling the Underground Market of Trading Mobile App Reviews”. In: *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2015-06. ISBN: 9781450336239. DOI: 10.1145/2766498.2766510 (cit. on p. 39).

[263] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. “Unauthorized Cross-App Resource Access on MAC OS X and iOS”. In: *arXiv.org* (2015), pp. 1–29. arXiv: arXiv:1505.06836v1 (cit. on p. 42).

[264] Luyi Xing, Xiaorui Pan, R Wang, Kan Yuan, and XF Wang. “Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating”. In: *IEEE Security and Privacy* (2014) (cit. on p. 42).

[265] Rubin Xu. “Improving application trustworthiness on stock Android”. PhD. University of Cambridge, 2015 (cit. on p. 87).

[266] Rubin Xu, Hassen Saıdi, and Ross Anderson. “Aurasium: Practical Policy Enforcement for Android Applications”. In: *USENIX conference on Security symposium* (2012) (cit. on p. 46).

[267] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *IEEE Symposium on Security and Privacy* 53.1 (2009), pp. 79–93. ISSN: 10816011. DOI: 10.1109/SP.2009.25 (cit. on p. 46).

[268] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. “Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs”. In: *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014. ISBN: 9781450329576. DOI: 10.1145/2660267.2660359 (cit. on p. 44).

[269] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. “Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis”. In: *ACM Conference on Computer and Communications Security (CCS)*. Berlin, Germany: ACM, 2013-11, pp. 611–622. ISBN: 9781450324779. DOI: 10.1145/2508859.2516689 (cit. on p. 43).

[270] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. “The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations”. In: *IEEE Security & Privacy* (2014). DOI: 10.1109/SP.2014.33 (cit. on p. 41).

[271] Yajin Zhou and Xuxian Jiang. “Dissecting Android Malware: Characterization and Evolution”. In: *IEEE Symposium on Security and Privacy* (2012-05), pp. 95–109. ISSN: 10816011. DOI: 10.1109/SP.2012.16 (cit. on pp. 44, 73).

[272] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. “Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets”. In: *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, 2012-02 (cit. on pp. 44, 51, 73).

[273] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. “Hybrid User-level Sandboxing of Third-party Android Apps”. In: *ACM Symposium on Information, Computer and Communications Security (Asia CCS)*. Singapore: ACM, 2015-04. ISBN: 9781450332453. DOI: 10.1145/2714576.2714598 (cit. on p. 46).

APPENDIX A

EXTRA INFORMATION

A.1 Number of computers

Table A.1 records the number of computers in use, population and derived computers per person data which I have gathered from various sources. Worldwide and US computers-in-use data for 2010-1980 came from eTForecasts [96]. Data for 1965 and 1955 came from a description of the history of the personal computer [132] and the estimate for 1950 is my best guess. World population data for 2010-1996 came from internet live stats [139] older data came from Wikipedia [255]. US population data came from the US Census Bureau [41, 40]

A.2 Lifetime of computers

Working out the average lifetime of computers is difficult as good records and statistics are not kept. The two methods used to estimate the lifetimes were: Extract the dates of first use and decommissioning of individual computers from historical records (§A.2.1). Use the estimated number of computers sold and installed base figures to estimate how long they must have lasted on average (§A.2.2, §A.2.3). One possible flaw with this latter strategy is that the installed base may have been calculated from the estimated sales figures using an assumed device lifetime [97]. If so this calculation would only re-derive an existing assumption. Unfortunately, the method used to estimate the installed base in the source data used is unknown.

Year:	2010	2005	2000	1996	1990	1985	1980
Computers-in-use	1.5×10^9	9.5×10^8	5.52×10^8	2.38×10^8	1.05×10^8	3.6×10^7	4.8×10^6
” in US	3.8×10^8	2.44×10^8	1.84×10^8	9.02×10^7	5.13×10^7	2.22×10^7	3.1×10^6
World population	6.9×10^9	6.5×10^9	6.1×10^9	5.8×10^9	5.2×10^9	4.8×10^9	4.4×10^9
US population	3.1×10^8	3.0×10^8	2.8×10^8	2.7×10^8	2.5×10^8	2.4×10^8	2.3×10^8
Computers per person	0.22	0.14	0.09	0.04	0.02	0.007	0.001
” in US	1.2	0.82	0.65	0.34	0.20	0.092	0.01

Year:	1965	1955	1950
Computers-in-use	20 000	250	5
World population	3.3×10^9	2.8×10^9	2.5×10^9
Computers per person	6×10^{-6}	9×10^{-8}	2×10^{-9}

Table A.1: Number of computers in use and population for the world and the US

Year	2010	2009	2008	2007	2006	2005
PCs sold	350 904 122	308 341 672	290 797 700	272 452 500	239 211 000	218 625 000
PCs in use	1 388 741 000	1 246 100 000	1 124 833 000	997 022 000	893 315 000	801 851 000
PCs replaced	208 263 122	187 074 672	162 986 700	168 745 500	147 747 000	
Fraction of PCs replaced	0.17	0.17	0.16	0.19	0.18	

Table A.2: PCs sold, in use and replaced each year

Year	2014	2013	2012	2011	2010	2009	2008
Smartphones sold (000s)	1 283 500	990 000	680 080	472 890	296 646	172 376	139 288
Smartphones in use (000s)	2 100 000	1 457 000	1 031 000	687 000	431 000	304 000	237 000
Smartphones replaced (000s)	640 500	564 000	336 080	216 890	169 647	105 376	
Fraction of smartphones replaced	0.44	0.55	0.49	0.50	0.56	0.44	

Table A.3: Smartphones sold, in use and replaced each year

	Mainframes	From	To	Lifetime
Cambridge	EDSAC	1949	1958	9
	EDSAC 2	1958	1965	7
	TITAN	1964	1973	9
	IBM 370/165	1971	1982	11
	IBM 3081D	1982	1989	7
	IBM 3084A	1989	1995	6
Oxford	Mercury	1959	1965	6
	English Electric KDF 9	1965	1971	6
	ICL 1901A	1971	1981	10
	ICL 2980	1977	1982	5
Manchester	Ferranti Mark 1	1954	1958	4
	Ferranti Mercury	1958	1963	5
	Ferranti Atlas	1962	1971	9
	MU5	1974	1982	8
	Summary	1949	1995	7.3 ± 2.1

Table A.4: Lifetimes of university mainframes

A.2.1 Lifetime of mainframes

The lifetime of mainframes can be estimated by sampling historical records. Table A.4 shows records from the universities of Cambridge [142], Oxford [148] and Manchester and gives an mean lifetime of lifetime of 7.3 ± 2.1.

A.2.2 Lifetime of PCs

Table A.2 shows the number of PCs sold, in use and replaced each year. The average proportion of PCs replaced from 2007-2010 was 0.17 ± 0.01, hence, the average lifetime was 5.83 ± 0.43.

PC sales figures for 2005 [189], 2006 [190], 2007 [194], 2008 [195], 2009 and 2010 [191] are from Gartner. PCs in use figures from Statsia [220, 221].

A.2.3 Lifetime of smartphones

Table A.3 shows the number of smartphones sold, in use and replaced each year. The average proportion of smartphones replaced from 2010–2014 was 0.51 ± 0.05, hence, the average lifetime was 1.97 ± 0.2.

Smartphone sales figures for 2013 and 2014 came from Strategy Analyt-

ics [166], for 2011 and 2012 from Statsia [222], for 2009 and 2010 [192] and 2008 [193] from Gartner. Smartphones in use figures for 2014 came from KPCB [168], for 2008–2013 from Statsia [223].

A.3 Vulnerabilities

Figures A.1, A.2, A.4 and A.5 show the proportion of devices in Device Analyzer exposed to different critical vulnerabilities. This displays the same data as shown in Figure 4.3.

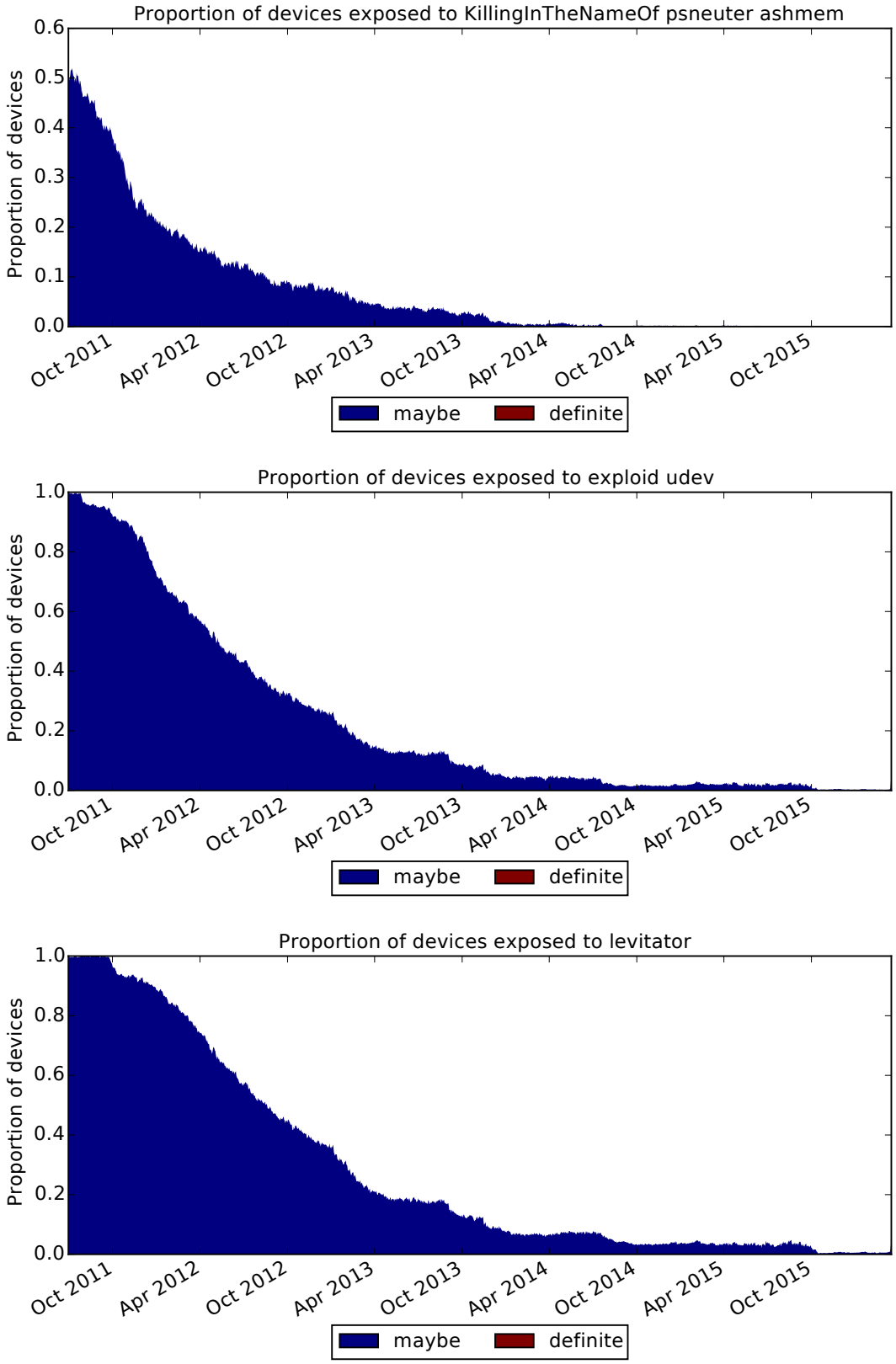


Figure A.1: Proportion of devices exposed to KillingInTheNameOf, exploid udev and levitator vulnerabilities.

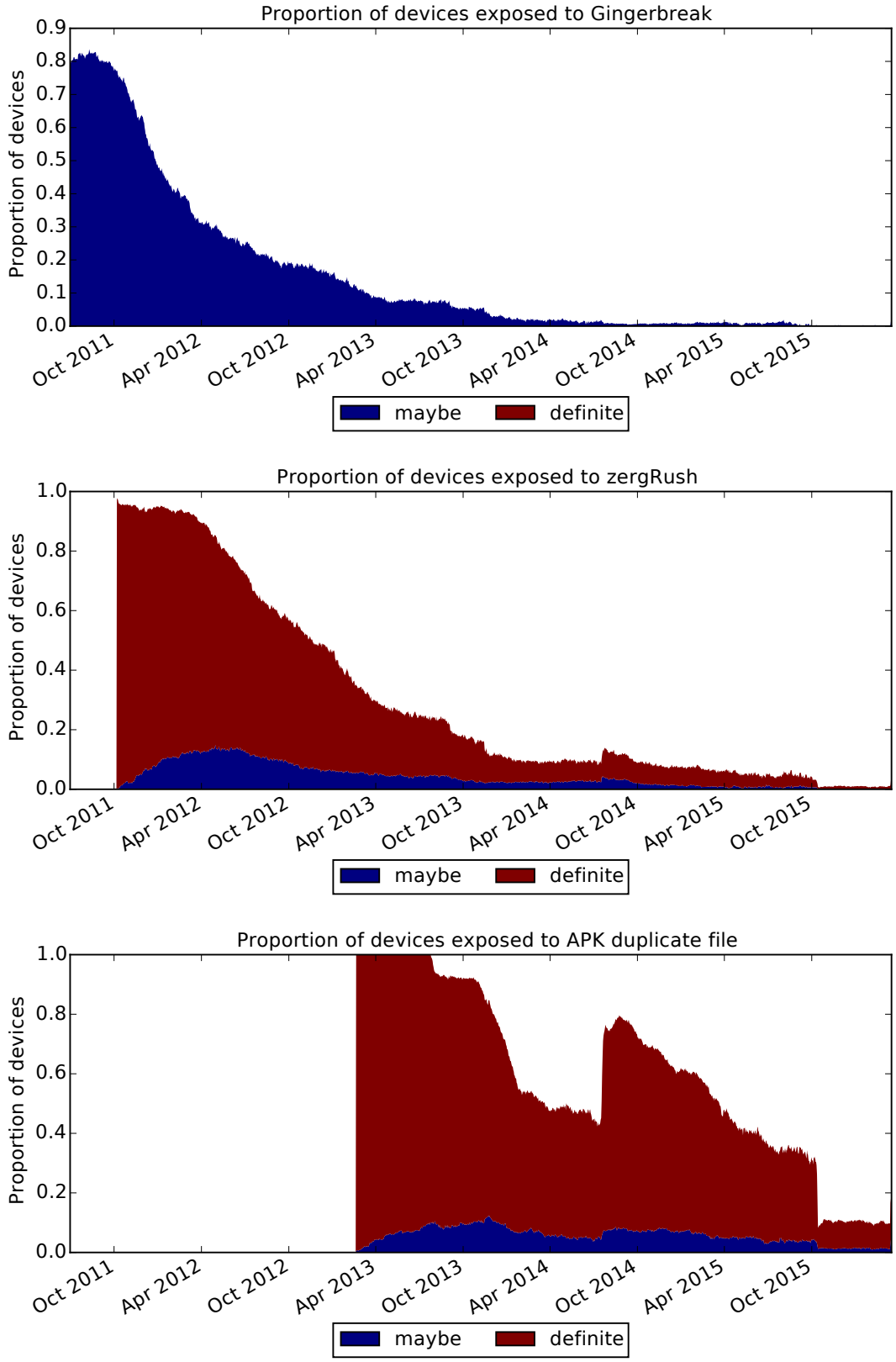


Figure A.2: Proportion of devices exposed to Gingerbreak, zergRush and APK duplicate file vulnerabilities.

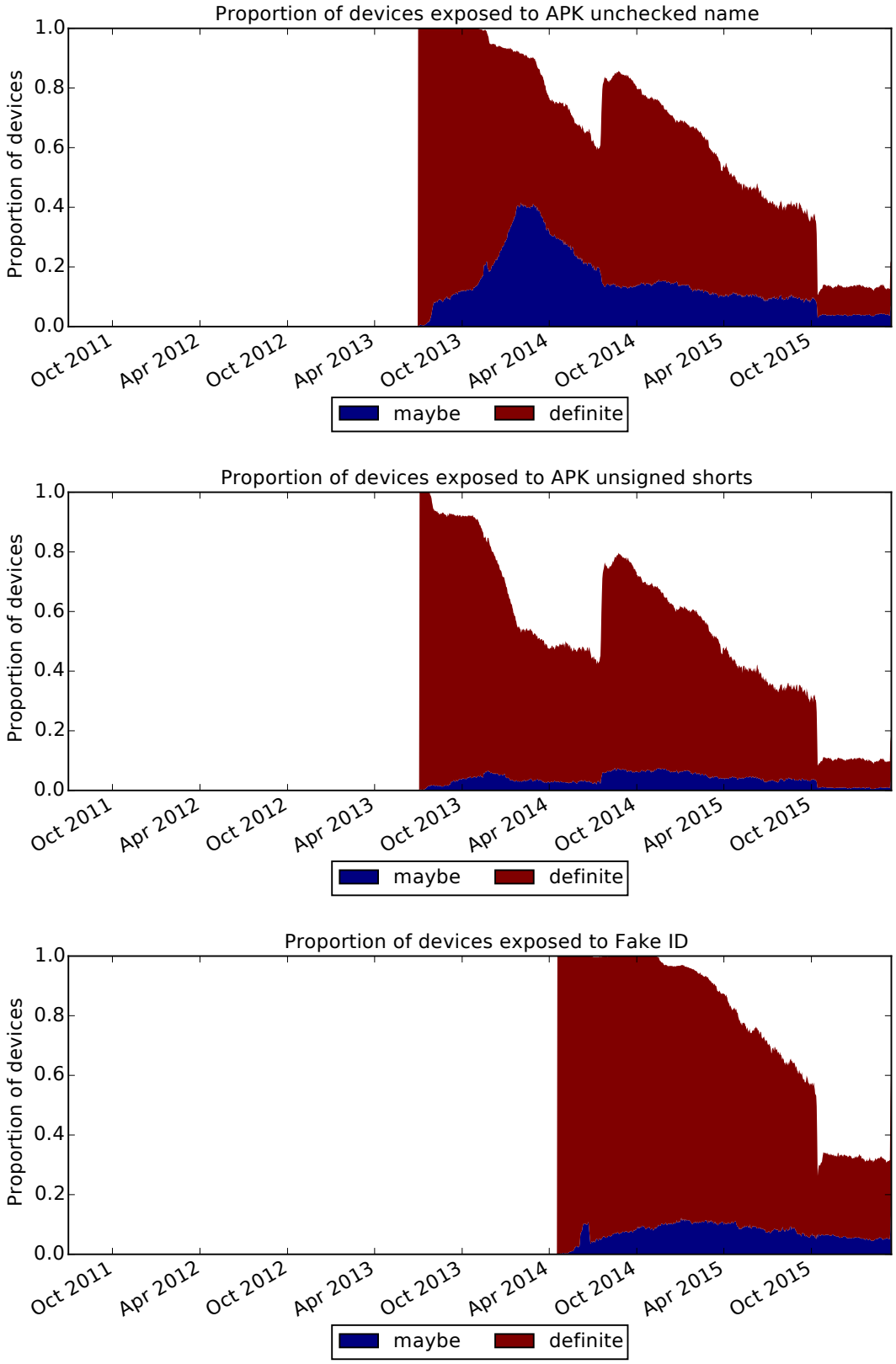


Figure A.3: Proportion of devices exposed to APK unchecked name, APK unsigned shorts and Fake ID vulnerabilities.

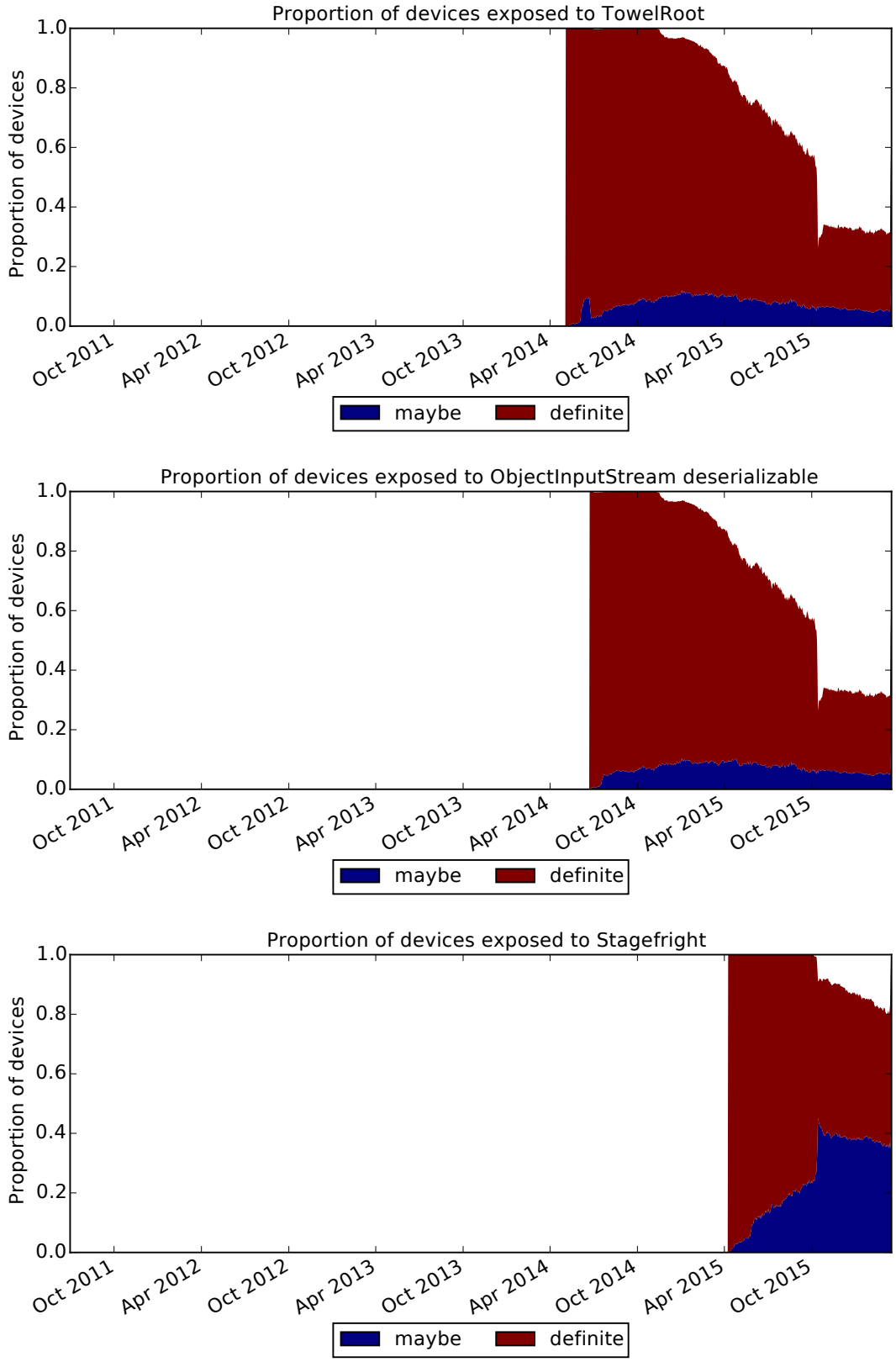


Figure A.4: Proportion of devices exposed to TowelRoot, ObjectInputStream and Stagefright vulnerabilities.

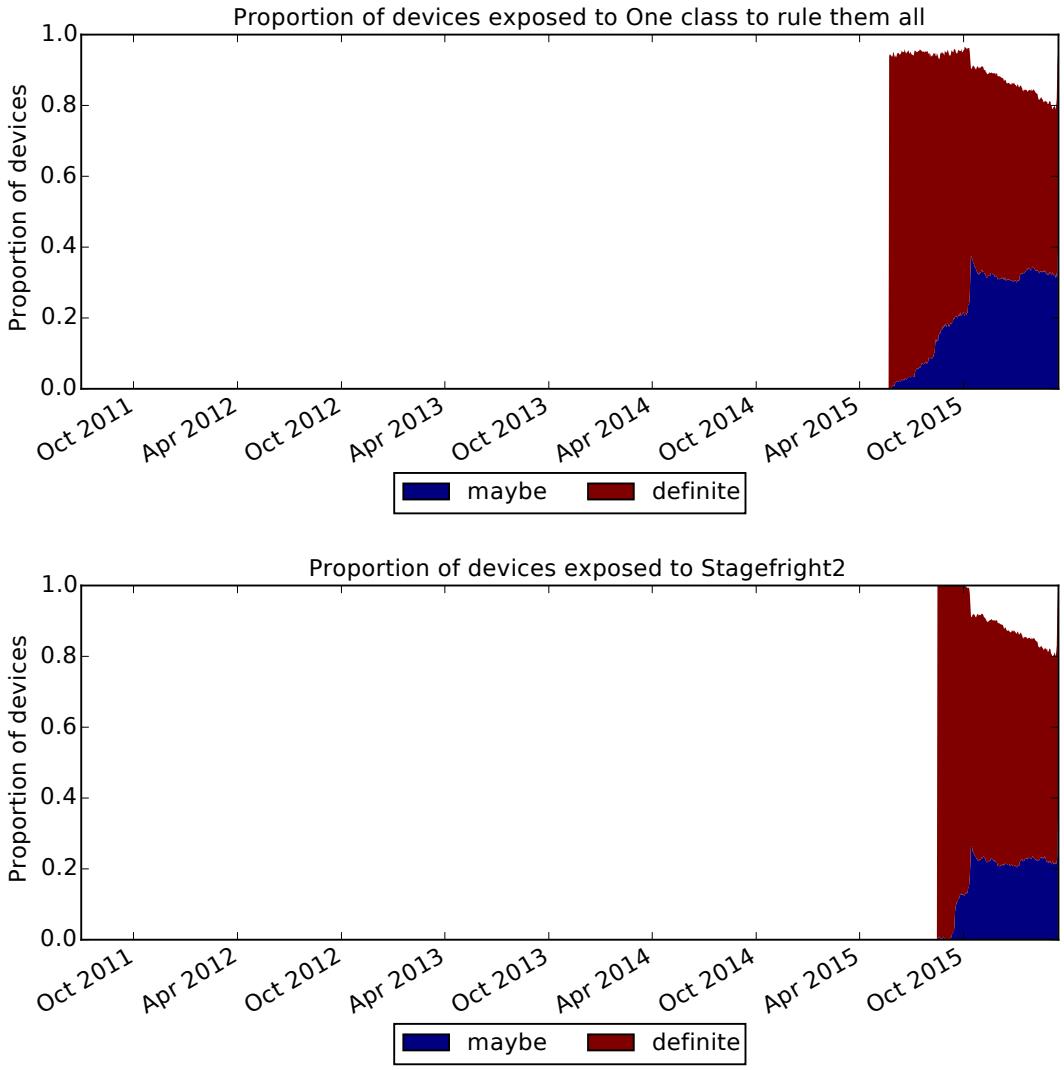


Figure A.5: Proportion of devices exposed to One class to rule them all, and Stagefright2 vulnerabilities.