Investigating Weaknesses in Android Certificate Security*

Daniel E. Krych†, Stephen Lange-Maney†, Patrick McDaniel†, William Glodek*

†Department of Computer Science and Engineering
The Pennsylvania State University
*US Army Research Laboratory, Adelphi, Marlyand

ABSTRACT

Android's application market relies on secure certificate generation to establish trust between applications and their users; yet, cryptography is often not a priority for application developers and many fail to take the necessary security precautions. Indeed, there is cause for concern: several recent high-profile studies have observed a pervasive lack of entropy on Web-systems leading to the factorization of private keys.[1] Sufficient entropy, or randomness, is essential to generate secure key pairs and combat predictable key generation. In this paper, we analyze the security of Android certificates. We investigate the entropy present in 550,000 Android application certificates using the Quasilinear GCD finding algorithm.[1] Our results show that while the lack of entropy does not appear to be as ubiquitous in the mobile markets as on Web-systems, there is substantial reuse of certificatesonly one third of the certificates in our dataset were unique. In other words, we find that organizations frequently reuse certificates for different applications. While such a practice is acceptable under Google's specifications for a single developer, we find that in some cases the same certificates are used for a myriad of developers, potentially compromising Android's intended trust relationships. Further, we observed duplicate certificates being used by both malicious and non-malicious applications. The top 3 repeated certificates present in our dataset accounted for a total of 11,438 separate APKs. Of these applications, 451, or roughly 4%, were identified as malicious by antivirus services.

Keywords: Android, Android Security, Android Certificate, RSA

1. INTRODUCTION

The use of cryptographic keys to authenticate and protect data is prevalent in mobile markets. An Android application's certificate provides the Google Play Store with the identity of the developer of the application. The certificate is usually self-signed and the developer holds the cryptographic key used in the certificate of their application.[2] A variety of cryptographic algorithms are used to generate the public-private key pairs used in asymmetric cryptography, or public key cryptography. Two of the most common algorithms to generate such pairs are RSA and DSA. When these algorithms are properly utilized, decrypting a public-private key pair is very difficult for an adversary, as it would require solving a fundamentally difficult mathematical problem. In order to generate sufficiently secure key pairs, a significantly large amount of entropy, or randomness, is needed to resist predictable key generation. The public and private key pairs used for signing Android applications are typically created using the Java Development Kit's (JDK) `keytool` utility. Once the key pairs are obtained, the `jarsigner` tool, which is also a part of the JDK, can be used to sign the Android APK.[2] The `keytool` and `jarsigner` utilities can be used as command-line tools or accessed through the Android Developer Tools (ADT) Graphical User Interface plugin for Eclipse.[3]

*Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

Further author information: (Send correspondence to Daniel E. Krych)
† Daniel E. Krych: dek5156@cse.psu.edu
† Dr. Patrick McDaniel: mcdaniel@cse.psu.edu
∗ William Glodek: william.j.glodek.civ@mail.mil

Modeling and Simulation for Defense Systems and Applications X, edited by Eric J. Kelmelis,
Proc. of SPIE Vol. 9478, 947804 · © 2015 SPIE · CCC code: 0277-786X/15/$18
· doi: 10.1117/12.2177498

Despite the user's need for security, markets can only provide it in a superficial way. The copious amount of applications in mobile markets and the broad definition of security means that some malicious, and insecure applications will be present in the market.[4] We believe that weak key generation may be present in Android applications. Cryptography is not typically a priority for Android developers when creating applications. Enck et al. explain that developers often fail to take security precautions, for instance, it isn't uncommon for sensitive information to be written to Android's centralized logs. They also found cases where this information was broadcast to unprotected Inter-Process Communication. Such mistakes may be due to a developer's lack of security knowledge. Developers are not cryptographers.[5] Most developers do not know what occurs under the hood when an SDK such as Eclipse ADT generates a key and signs their application. Their focus is on generating an application that will provide some utility to an end user and not on properly using cryptography. We believe that a person with malicious intent could capitalize on these weaknesses, falsify a certificate, and produce or alter applications as a trusted developer.

The majority of downloads from the Google Play Store are made up of only a small percentage of free applications. Additionally, about 25% of the Google Play store is duplicative application content.[6] These free applications would be then be a main target for malicious misuse. We believe that the average user is aware of this duplicative content, and thus try their best to only download an application from the original developer. If weaknesses in key generation exist and these can be misused, malicious applications could be disguised as those published by these original, trusted developers. The average user would then download these applications without any inclination that they are now filled with malware.

Our main contributions include the analysis of a large dataset of Android certificates for weaknesses in key generation, and an investigation into the repeated use of certificates across applications and potential vulnerabilities associated with this reuse. We examined the most repeated certificates on the market and note applications signed with these that were marked as malicious by multiple antivirus softwares. Additionally, we explained the trends of increasing bit-sizes used for RSA encryption within Android certificates over time.

2. RELATED WORK

Heninger et al. explain the catastrophic failures of RSA and DSA when random number generators malfunction. After crawling the Internet, they pulled and tested over 5.8 million TLS certificates and 6.2 million SSH host keys. The private keys for approximately 64,000 (0.50%) of TLS hosts and 108,000 (1.06%) of SSH hosts were computed due to insecurities in encryption methods. Through their research, they found a lack of randomness on headless or embedded devices from their inability to pull data from a large enough entropy pool. Along with this, they found a boot-time entropy hole present in Linux's random number generator. Upon booting Linux, there was a time where insufficient entropy was present to generate a key. Additionally, as Android applications developed on a Linux system utilize `/dev/urandom` instead of the suggested `/dev/random`, the possibility exists that weaknesses may be present in these certificates.[1]

Heninger et al. used their "Quasilinear GCD finding" algorithm to discover key generation weaknesses. This algorithm was based off of Bernstein's work and Euclid's greatest common divisor algorithm.[7] In general, an RSA modulus, N, is calculated by multiplying two very large, randomly chosen prime numbers, `p` and `q` together. If two moduli share a common prime, the Greatest Common Divisor (GCD) of the two can be used to find the `p` and `q` values involved in the creation of the moduli. It then follows that the private key is easily computable. They began by investigating the fastest known factoring method of integers: the number field sieve. This method has heuristic complexity $O(2^{n^{1/3}(\log n)^{2/3}})$ for n-bit integers. It has not been publicly successful for breaking any integer over 768 bits. Using Euclid's algorithm they could calculate the GCD of two integers as if any two moduli share a prime factor, `p` or `q`, the GCD of both moduli would expose that common prime. This GCD could then be divided into the modulus to determine the second prime factor used in key generation. Using this method they could find the GCD of two 1024-bit RSA moduli in 15 μs, but this process would still take roughly 30 years with their data set.[1]

Their quasilinear-time algorithm takes in a collection of integers and outputs their co-primes using factorization. It computed weaknesses in their data set in 5.5 hours. The algorithm starts by using a product tree to compute the product of all the moduli, and then uses a remainder tree to reduce the product modulo each input.

Upon completion, the algorithm outputs the GCD of each of the inputted moduli with the product of each other moduli.[1] We utilized their algorithm and code in our testing for weaknesses in Android certificates using RSA.

Van Bockhaven and Gieske also worked on cracking weak keys present in Android applications during the same time period that we conducted our research. Their dataset included a total of 75,126 APKs while ours included 571,431 APKs, more than 7.5 times larger. Their results for weaknesses in RSA certificates agree with ours (see section 4.1). They did find three 512-bit RSA keys, most likely the same three that we discovered, and they factored two of them in a clustered effort. Following this, their attempts to upload changes to these applications, pretending to be the original developers, failed. This could mean that our hypothesis of taking over as the developer of an application may prove to be more complicated than initially thought. Their efforts help to reassure our results and research, as do ours to theirs.[8]

3. METHODOLOGY

To investigate whether the same weaknesses identified in SSH host keys and TLS certificates were present in the Android application ecosystem, we obtained a dataset of 571,431 Android applications (using RSA or DSA encryption). These applications were pulled from the market by Matt Dering over the course of 7 months in 2013, by crawling the Google play store with an application size cap set at 50 MB. This pool became our dataset for all of the following tests. On September 9th, 2013, the last start validity date for an application in our pool, the Google Play Store had a total of 953,300 applications. Thus, our dataset encompassed roughly 60% of the entire market at that time.[9] Extracting the certificates from these applications revealed 551,553 applications using RSA encryption and 19,878 using DSA encryption.

3.1 RSA Certificates

We tested for vulnerabilities in RSA certificates using the quasilinear-time GCD finding algorithm.[1] We first looked into the certificates using RSA encryption, and then those using DSA. We believed that by calculating the private keys of these applications and reproducing their certificates, we could sign a malicious application with the trusted certificate. To reproduce a certificate the encryption needs to be broken. RSA encryption breaks when the two large prime numbers involved in the key generation, `p` and `q`, are determined. Similarly, one way that DSA encryption can break is when an `r` or `s` value is shared by more than one DSA certificate.

During our investigation of the Quasilinear GCD finding algorithm, we also briefly researched the entropy involved in key generation for Android applications. We investigated possible sources of weak key generation. We first created an Android application using ADT and Eclipse, then signed it in debug mode through the Eclipse GUI.[3] By examining ADT's signing process we determined that the Android SDK build tools use the `keytool` utility to create keys and the `jarsigner` utility to sign the application's `.apk` file. Both utilities are included in the Java Development Kit (JDK).[2] We used the `keytool` utility along with Linux's `strace` tool in order to determine the system calls to the pseudorandom number generators found in `/dev/random` and `/dev/urandom` to better understand the entropy pool being created and used for key generation. While generating an RSA certificate with a 2048-bit modulus we logged 2 calls to `/dev/random` and 5 calls to `/dev/urandom`. Heninger et al. determined a boot-time entropy hole on headless or embedded devices that was a window of vulnerability during which Linux's `/urandom` may be entirely predictable.[1]

Using `openssl` we extracted the plaintext modulus from each certificate.[10] We then sorted the moduli according to their bit-size to be inputted into the `fastgcd.c` program, which uses the quasilinear-time GCD finding algorithm.[1] The bit-sizes in our dataset varied from 512-bit to 16384-bit. This range of bit-sizes was unexpected, given that the standard bit sizes are 1024-bit and 2048-bit. We also encountered odd cases in which the bit-size did not follow the uniform scale. For example, one modulus was 3072-bit. These odd moduli were not useful for our testing, and were excluded from our testing for vulnerabilities.

As seen in Figure 1, we discovered that 99.8% of the certificates were signed using keys that used 1024-bit or 2048-bit encryption. Additionally, three unique 512-bit RSA keys were discovered. These are known to be insecure, as larger keys sizes have been cracked. The 768-bit modulus (232-digit number) was broken on December 12, 2009 using the number field sieve (NFS).[11] We also found that surprisingly only 34% of 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli were unique. Note that our dataset did include multiple versions of some applications, which were updated during the pooling of Android applications.

Bit-Size	`fastgcd.c` run Time (seconds)	Moduli Count	Unique Moduli Count	Unique Moduli Percent
512-bit	0.001	4	3	75.0000%
1024-bit	106.855	348484	118442	33.9878%
2048-bit	136.704	202042	69255	34.2775%
4096-bit	0.311	826	282	34.1404%
8192-bit	0.019	52	18	34.6154%
16384-bit	N/A	2	1	50.0000%
Odd-bit	N/A	142	55	38.7324%

Figure 1. `fastgcd.c` **Run Times and Moduli Uniqueness**

3.2 DSA Certificates

We then performed a small investigation on the pool of 19,878 DSA certificates. As DSA encryption differs from RSA encryption, the Quasilinear GCD finding algorithm could not be used to determine weaknesses. One quick way of investigating potential weaknesses is to look into the DSA signature of the certificate. The DSA signature consists of a pair of values, (`r`, `s`), which are computed using several components of the public key, a randomly chosen ephemeral private key, and the hash of the message.[1] Similar to RSA modulus' `p` and `q` values, if two different DSA signatures share either an `r` or `s` value, the shared values lead to a calculable private key.

3.3 Certificate Reuse

In order to investigate the amount of certificate reuse, we began by finding the top ten most common moduli. Our results showed that every certificate with a matching modulus had exactly the same certificate. Interestingly, the most frequently occurring modulus was repeated 4,515 times. A typical developer would not create that many applications or updates to applications. We looked at the certificates attached to the most often repeated moduli and found that, in each case, the "Issuer" was a company which enables a developer to create an application for free, with no coding required; consequently, any developer who creates an Android application through this site is published with the same certificate. This practice explains why some moduli were repeated many times.

4. RESULTS

4.1 RSA Certificates

To test the weaknesses present in the Android Market, we addressed our dataset of 551,553 applications using RSA encryption. As explained earlier, we parsed these and collected the respective moduli to use as our input into the Quasilinear GCD finding algorithm. The program `fastgcd.c`, which was written by Heninger et al. and utilizes this algorithm, was open source and freely available online †.[1] After organizing the moduli in our dataset according to bit-size, we found 348,484 1024-bit moduli and 202,042 2048-bit moduli, as the most prevalent sizes from the RSA certificates pool.

We ran our tests in an `Ubuntu` 12.04 virtual machine (4 GB RAM allotted) on a MacBook pro with a quad core 2.6 GHz Intel Core i7 processor. To our surprise, we found that a large number of the moduli were repeated numerous times. Of the 1024-bit dataset, only 118,422 moduli were unique, giving us 33.982% uniqueness. From the 2048-bit dataset only 69,255 moduli were unique, giving us 34.277% uniqueness. For a full list of unique moduli versus all the moduli, refer to Figure 1.

We sorted and parsed the unique moduli of each bit-size, then used this as our input for the `fastgcd.c` program. In 106.855 seconds the 1024-bit dataset outputted a total of 0 weak moduli. Similarly, in 136.704 seconds the 2048-bit dataset revealed a total of 0 weak moduli. The results were the same for all datasets

†`https://factorable.net`

regardless of the bit-size of the moduli. Refer to Figure 1 to see a complete list of our times. Each run incorporated all of the moduli of the associated bit-size. The 16,384-bit and odd-bit moduli results are not available, as there were no matching bit-sizes to compare. Our findings did not support our original hypothesis that weaknesses exist within the random number generation for Android certificate key pairs.

4.2 DSA Certificates

Of the 19,878 DSA signatures, we found that only 6,001, approximately 30% were unique. Recall that RSA certificates were only 34% unique. Additionally, a few of these DSA certificates seemed to not have `r` or `s` values. Upon further investigation we found that one was actually RSA encryption in a `.DSA` file, and two others used `dsaEncryption` for their Signature Algorithm with `x509v3` extensions instead of `dsaWithSHA1` which the rest used as their Signature Algorithm.

4.3 Certificate Reuse

After finding that the most frequently occurring modulus was used by a company, which boasts that no coding is required to publish an application and shares its certificates, we decided to investigate whether any malicious applications have been created using this website. The top 3 certificates/moduli seen in our RSA certificate dataset appear a combined amount of 11,438 times in different APKs. Using virusTotal [‡] we determined the malicious rating of these applications through the use of multiple antiviruses. A total of 3,345, or roughly 29%, of these applications were not found in the database. Still, of those present in the database, 451 were marked as malicious by 2 or more antivirus softwares, accounting for approximately 4%. The most seen modulus had only one APK that was tagged as malicious, and was marked by 2/50 of the antivirus softwares used for testing. The second most seen modulus accounted for the other 450 APKs that were tagged as malicious by 2 or more of the antivirus softwares. It appears that the second company is much more lenient in their approval of applications. The highest malicious score seen was 11/52. We further investigated ten of the antiviruses. The antivirus `TrendMicro-HouseCall` detected several variants of the `TROJ_GEN.F47V0` and the antivirus `VIPRE` detected the `Trojan.AndroidOS.Generic.A` and `Adware.AndroidOS.RevMob.a`.

According to Google's Android developer website, every application installed on the Android system must have a certificate which was digitally signed using a private key held by the application's developer. The certificate is used by the Android system to identify the author of an application, and for, "Establishing trust relationships between applications".[2] Thus, when several developers share the same certificate for their separate applications, they may be creating weaknesses that can lead to malicious misuse. If a malicious application uses a shared certificate, we believe it may also pose a threat to all the other applications sharing said certificate.

5. DISCUSSION

At this point in our research we had received our most crucial results, that 0 vulnerabilities were present in our dataset according to the methods used by Heninger et al.[1] We had also gained a better understanding of why there were so many repeated moduli in our dataset. This was due to the fact that some certificates were being shared by a variety of developers. This included websites that provide templates for creating Android applications online, and when publishing your application, used the same certificate (or set of certificates) for each customer. The sharing of certificates could be a vulnerability in itself.

The post-analysis of our results led us to reconsider our dataset as a whole. In total, we tested 571,431 Android applications, although there are roughly 1.5 million applications on the market at this time.[9] One reason for our reconsideration is the constantly increasing market; our dataset was collected in 2013 when there were fewer applications on the market. Another reason is that the process of crawling the Google Play Store was only intended to collect applications under 50 MB. After reviewing our results, we theorized as to why we found zero weaknesses. Comparing our results to those found by Heninger et al., we note that their results were primarily consequences of generating keys on headless or embedded devices, which were unable to generate enough entropy. We thus conclude that, generally, the issue of sufficient entropy generation is not a cause for concern in systems which are used to sign Android applications certificates. They hypothesized this and explained that for RSA and DSA keys generated on traditional PCs, they had no reason to doubt their security.[1]

[‡] `https://www.virustotal.com`

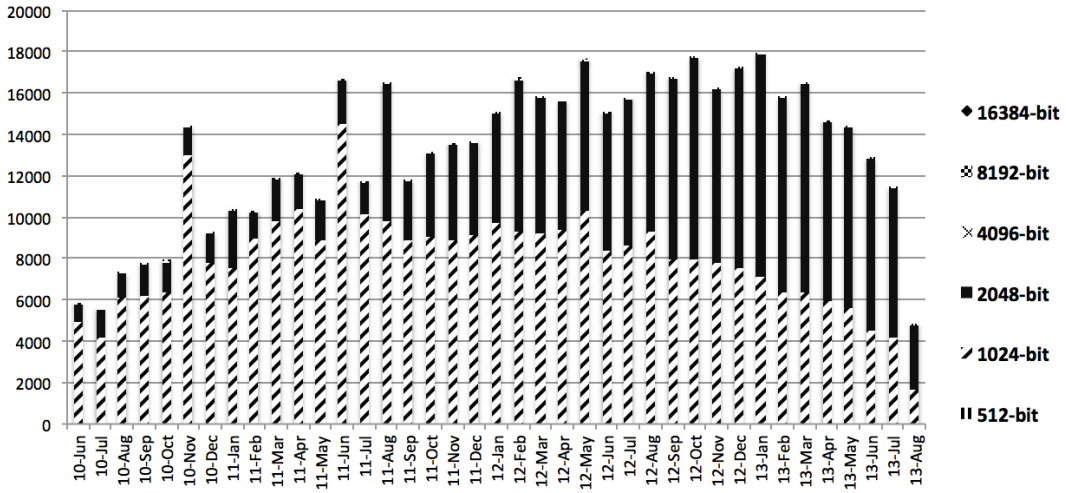

Figure 2. **RSA Certificates - Moduli Bit-sizes Used Over Time**

Not wanting our dataset go to waste, we decided to take time to analyze the Android certificates and their attributes. Given the wide range of bit sizes, 512-bit to 16384-bit, with 55 unique odd-bit sizes such as 2100-bit, 3072-bit, 4004-bit, and 5120-bit scattered throughout, we decided to look into this non-uniformity further. We also wanted to determine when 2048-bit became more prevalent than 1024-bit for encryption. Additionally, we wondered if by identifying trends of developers switching to higher bit-sizes for encryption, we would be able to associate these to security vulnerabilities being discovered. We assumed that over time the bit-sizes would increase as developers would slowly increase their moduli to ensure that their encryption remain difficult to break. We used the start date of the validity for the certificate as a rough idea of when it had been created.

Using Figure 2, we were able to determine that Sep-2012 was the tipping point from 1024-bit to 2048-bit moduli. It is clear that 2048-bit moduli use increased each year, and that after September 2012 it was the most commonly used bit-size. It is important to note: 2013 does not have complete results as the Android applications were pulled from the Google Play Store some of August 2013 and some up until September 9, 2013. We excluded these months for this purpose, but can roughly project a total of 110,000 2048-bit moduli and 70,000 1024-bit moduli used in 2013. Additionally, we cut down the scope of our graph to begin at June 2010, as this is when 2048-bit moduli were on the rise. According to our collected data, in April 2010 there were 4 2048-bit moduli, in May 2010 we observed 71 2048-bit moduli, and in June 2010 it jumped to 824 2048-bit moduli. Finally, in September 2012, 2048-bit moduli became more prevalent than 1024-bit with a count of 8741 to 7989, respectively. In addition to excluding dates for better representation, we also had to exclude some anomalies of certificates with validity beginning in 1952 through 1956, 1992, and the early 2000's, as developers had altered these. We excluded these outliers from our dataset.

We then investigated the patterns of 2048-bit modulus use and attempted to map events in the security world to the increased use of these more secure encryption methods. First we looked at June 2010, since from May to June 2048-bit encryption methods increased over 1000%. One possible cause for the shift from 1024-bit encryption to 2048-bit encryption is the factorization of the 768-bit modulus (232-digit number) on December 12, 2009 using the number field sieve (NFS). Kleinjung et al. released their work in January of 2010, and five months later we see this dramatic increase in the use of 2048-bit encryption. They even state that compared to a 768-bit RSA modulus the factorization of a 1024-bit one would be several thousand times harder and for a 512-bit modulus, several thousand times easier. They also point out that as the first factorization of a 512-bit RSA modulus was done in 1999, phasing out 1024-bit RSA moduli would be necessary over the next three to four years.[11] We then looked at August 2011, when there was around a 500% increase. We did observe that this was the month when the most commonly repeated certificate's validity began, which uses 2048-bit encryption. As this certificate was found over 4,500 times in our data set, there is a good chance that this certificate explains this spike. This also means that this spike is irrelevant to our data since these applications could have been

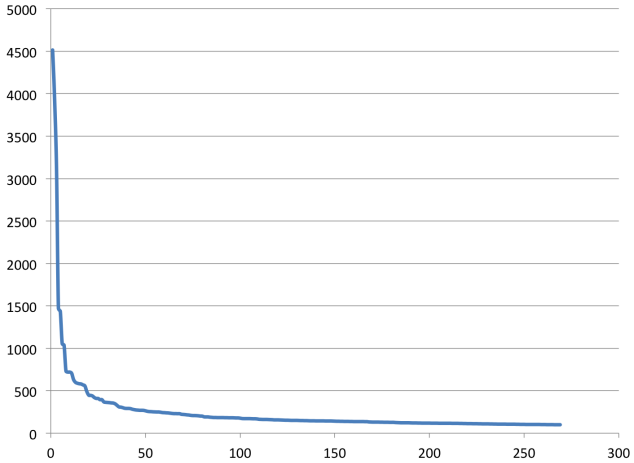

Figure 3. **Number of times Repeated vs. Most to Least Repeated RSA Modulus**

created and uploaded anytime between August 2011 and the last day an application in our dataset was pulled in September 2013. We then tried to find reasoning behind the tipping point for 2048-bit use in September 2012, but believe this was just caused by the gradual incline and not an event during that exact time period.

In Figure 3, we observed the top repeated RSA moduli, showing the number of times they were repeated versus the most to least repeated moduli. This graph shows all repeated moduli that were seen 100 or more times. Roughly twenty moduli were repeated over 500 times, and about seventy-five repeated over 250 times. The graph ends at the 269th most repeated moduli, which occurs exactly 100 times. We selected this span of data to graph in order to be able to view the curve. The majority of the curve occurs within this span and trails off for the 187,767 remaining moduli. This sharp decrease in repeated moduli tells us that across the Android Market as a whole, a moduli being repeated over 100 times is uncommon.

6. FUTURE WORK

We can see our research being continued and adapted in multiple ways. Some of these include investigating third-party markets, determining if weaknesses arise when applications share certificates, determining whether DSA is biased, and of course, retesting the android market.

Applying this research to third-party markets may yield weaknesses, as these markets are less strict with application acceptance than the Google play store.

During our research we found an alarmingly high number of certificates that were repeated across applications. As Google's Android developer website explains, an application installed on an Android system must have a certificate which was digitally signed using a private key held by the application's developer. The certificate identifies the application's author and is used for establishing trust relationships across applications.[2] We discovered several websites that provide a service for creating android applications with little to no coding involved; these sites had a high number of repeated certificates. According to our data, one site in particular used the same certificate 4,515 times across different applications. This means that different developers share the same certificate. This could potentially compromise this trust relationship. Even if the trust relationships between applications sharing a certificate yield no additional risks, the sharing still makes these certificates bigger targets. By compromising these shared certificates, an adversary will be able to do more damage with less effort. Cracking RSA on a certificate used across 4,515 applications is much more detrimental than that of a certificate used across a few applications.

Our work could be reapplied in the future to pull the latest android applications from the Google play store and retest for weaknesses. Heninger et al. only found that about 1% of their data showed weaknesses.[1] In comparison, our dataset may have been too small. As was previously stated, we determined that our dataset encompassed roughly 60% of the market at the time the applications were pulled. There are now approximately

1.5 million android applications on the Google play store, compared to about 950,000 applications present when we finished pulling applications.[9]

Further research into DSA certificates could reveal weaknesses in DSA, but as only 3.6% of our application pool contained APKs using DSA encryption, we focused on RSA certificates. We quickly checked for any obvious weaknesses by analyzing the DSA signatures to see if any different signatures shared a common `r` or `s` value, but found zero cases of shared values.

7. CONCLUSION

We have found the Google play store's applications to be unbiased in terms of weaknesses in random number generation. We used the Quasilinear GCD finding algorithm created by Heninger et al. to test a large pool of android applications for weak or biased keys. Using this method we determined that there were no weaknesses in our dataset. It is important to note that the boot-time entropy hole is not prevalent to generating android certificates; certificates are generated well after a computer is booted, while in their work the ssh key was generated almost immediately upon first boot. The testing done by them refers to headless or embedded devices which commonly generate keys automatically on first boot. They explain that compared to traditional PCs, these devices may have limited entropy sources. Our results support the theory that any android certificate will be generated on a system with sufficient entropy.[1]

We also investigated Android's certificate signing process to find what entropy pools were used to generate a key by tracing the system calls. We used eclipse ADT to create and sign android applications.[3] Furthermore, we used `keytool` on the command-line in joint with `strace` to observe any system calls to `/dev/random` or `/dev/urandom`.[2] While generating an RSA certificate with a 2048-bit modulus we logged 2 calls to `/dev/random` and 5 calls to `/dev/urandom`. Both of these prngs are being used, but by the time ADT calls upon them, their entropy pools provide sufficient randomness.[1] Additionally, we used our dataset to better understand how the applications on the android market, their certificates, and their encryption methods have changed over time. From this research, we were able to hypothesize future work and potential weaknesses. Some of our future work suggests re-testing the android market as additional applications become available.

Additionally, we discovered that organizations were frequently reusing certificates across applications. Certain certificates were even used across a myriad of developers. We hypothesize that this could potentially compromise Android's intended trust relationships between applications. Malicious applications were discovered that shared certificates with ones that were non-malicious. We analyzed the top 3 repeated certificates present in our dataset. These accounted for a total of 11,438 separate APKs. Approximately 4%, or 451 of these were identified as malicious by antivirus services. If these trust relationships could be compromised, then applications sharing certificates with malicious applications may pose a threat to Android security.

8. ACKNOWLEDGMENTS

We would like to thank Joshua Edwards for his programming expertise, aiding in the design of several of our programs and the mitigation of multiple programming bugs. We would also like to thank Matt Dering for providing us with our Android application dataset. Additionally, we would like to thank Dr. Robert Walls for his helpful comments.

REFERENCES

[1] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A., "Mining your ps and qs: Detection of widespread weak keys in network devices," in [*Proceedings of the 21st USENIX Conference on Security Symposium*], *Security'12*, 35–35, USENIX Association, Berkeley, CA, USA (2012).

[2] "Android app signing." http://developer.android.com/tools/publishing/app-signing.html.

[3] Developers, A., "Adt plugin for eclipse," *URI: http://devel-oper. android. com/sdk/eclipse-adt. html* (2012).

[4] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S., "A study of android application security.," in [*USENIX security symposium*], **2**, 2 (2011).

[5] Viega, J. and McGraw, G., "Building secure software: How to avoid security problems the right way (paperback)," (2011).
[6] Viennot, N., Garcia, E., and Nieh, J., "A measurement study of google play," in [*The 2014 ACM international conference on Measurement and modeling of computer systems*], 221–233, ACM (2014).
[7] Bernstein, D. J., "How to find smooth parts of integers," *URL: http://cr. yp. to/papers. html# smoothparts. ID 201a045d5bb24f43f0bd0d97fcf5355a. Citations in this document* **20** (2004).
[8] Van Bockhaven, C. and Gieske, S., "Weak key cracking of android applications," (2014).
[9] "Number of android applications." http://www.appbrain.com/stats/android-app-ratings.
[10] Cox, M., Engelschall, R., Henson, S., Laurie, B., et al., "The openssl project," (2002).
[11] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., et al., "Factorization of a 768-bit rsa modulus," in [*Advances in Cryptology–CRYPTO 2010*], 333–350, Springer (2010).