

Spring 2014

Automating NFC Message Sending for Good and Evil

Nikki Benecke Brandt
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Computer Sciences Commons](#)

Recommended Citation

Brandt, Nikki Benecke, "Automating NFC Message Sending for Good and Evil" (2014). *Master's Projects*. 419.
DOI: <https://doi.org/10.31979/etd.hxe2-aafh>
https://scholarworks.sjsu.edu/etd_projects/419

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Automating NFC Message Sending for Good and Evil

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Nikki Benecke Brandt

May 2014

© 2014

Nikki Benecke Brandt

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Automating NFC Message Sending for Good and Evil

by

Nikki Benecke Brandt

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2014

Mark Stamp Department of Computer Science

Melody Moh Department of Computer Science

Thomas Austin Department of Computer Science

ABSTRACT

Automating NFC Message Sending for Good and Evil

by Nikki Benecke Brandt

Near Field Communication (NFC) is an emerging proximity wireless technology used for triggering automatic interactions between mobile devices. In standard NFC usage, one message is sent per device contact, then the devices must be physically separated and brought together again. In this paper, we present a mechanism for automatically sending multiple messages without any need to physically decouple the devices. After an introduction to NFC and related security issues, we discuss the motivation for—and an implementation of—an automation framework for sending repeated NFC messages without any need for human interaction. Then we consider how such an automated mechanism can be used for both a denial of service attack and as a platform for fuzz testing. We present experimental evidence on the efficacy of automated NFC as a vector for achieving these goals. We conclude with suggestions for future work and provide some overall insights.

ACKNOWLEDGMENTS

Thank you to Dr. Mark Stamp for maintaining confidence in my project and for laughing along with my many mishaps.

Thank you to Josh, Grey, and Wren Brandt for giving me a reason to stop working occasionally, and for only very rarely complaining about the eternal buzzing noise.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Background	3
2.1	Near Field Communication	3
2.1.1	NDEF Format	4
2.2	Related Work	5
2.2.1	NFC Security	5
2.2.2	Denial of Service in Mobile Devices	8
2.3	NFC on Android	9
2.4	Fuzz Testing and NFC	10
2.4.1	Fuzz Testing	10
2.4.2	NFC-Specific Fuzzing	12
3	Motivation for Automation	15
3.1	Comparison to Existing Solutions	16
3.2	The Relationship Between Denial of Service and Fuzz Testing	17
4	Automating NFC messaging	20
4.1	An Ideal Automated NFC Application	20
4.2	Automated NFC Application Design	22
4.3	The Application	22
4.3.1	Interface Design	22
4.3.2	Backend	25

4.4	Barriers Preventing Automation	30
4.4.1	Getting Permissions	31
4.4.2	Pacifying the SDK	32
4.4.3	Installing as a System Application	33
4.4.4	Unlocking and Rooting the Phone	34
4.4.5	Using Reflection to Access <i>setP2pModes</i>	35
4.4.6	Disabling the Touch to Beam interface	36
4.4.7	Periodically Resetting NFC	38
4.5	Logging	38
5	Using Automation for a Denial of Service Attack	41
5.1	Entities	41
5.2	Interactions	42
5.3	Attack Assumptions	43
5.3.1	How Close is Close Enough?	44
5.4	Attack Implementation	45
5.5	Results	47
5.5.1	Varying file size	49
5.5.2	Varying test length	51
5.6	Discussion	53
6	Using Automation for Fuzz Testing	63
6.1	Entities	63
6.2	Interactions	64
6.3	Assumptions	65

6.4	Generating Random Inputs	66
6.5	Fuzz Test Implementation	66
6.6	Initial Results	68
6.7	Discussion	69
7	Future Work	70
8	Conclusions	74
 APPENDIX		
A	Practicalities	79
A.1	Physical Phone Setup for Experiments	79
A.2	Rooting the Galaxy Nexus	79
A.3	Using dd to Generate Files with Random Contents	81
A.4	Input File Format	81
A.5	Interesting Failures that Occurred During Testing	82
B	Additional graphs	86
B.1	Storage space change for various file sizes	86

LIST OF TABLES

1	NDEF record fields	6
2	Content source options	25
3	Test length options	25
4	Relevant NFCfunctions	29
5	Necessary permissions for the automation app	31
6	Solutions to problems	33
7	Values logged	40
8	Type-Name-Format (TNF) values	67
9	Available NDEF types	67

LIST OF FIGURES

1	NDEF message containing multiple NDEF records	5
2	Example NFC transaction	5
3	Comparison of denial of service and fuzz testing	19
4	Automated NFC application user interface	23
5	Disabling the Lint warning	33
6	Installing the application as a system application	34
7	Touch to Beam UI [41]	37
8	Performance of NFC at various distances	45
9	Idle vs active	48
10	Battery drain over extended time period	49
11	Battery drain for various file sizes	50
12	Storage consumption over 10 trials for file size=50kb	52
13	Average storage consumption for file size = 50kb	53
14	Storage consumption over 10 trials for file size=500kb	54
15	Average storage consumption for file size = 500kb	55
16	Storage consumption over 10 trials for file size=5Mb	56
17	Average storage consumption for file size = 5Mb	57
18	Number of NDEFs sent for various file sizes	58
19	Victim phone's network usage for file size = 5MB	59
20	Battery drain versus attack length	60

21	Battery temperature before and after 30 minutes of denial of service attack	61
22	Android notifications screen	62
A.23	Nexus Root Toolkit UI	80
A.24	Too many notifications	84
A.25	Browser Hang	85
B.26	Storage space change for file size=50kb	87
B.27	Storage space change for file size=500kb	88
B.28	Storage space change for file size=5mb	89

CHAPTER 1

Introduction

In the past several years, Near Field Communication (NFC) has started to appear on a large number of mobile devices. By the end of 2013, an estimated 1 in 3 smartphones being sold had NFC capabilities [6]. Proponents of NFC promise an appealing user experience that allows users to easily perform tasks such as file transfers or contactless payments [32]. A broad variety of applications for NFC have appeared over the past decade. Modern applications for NFC include transit system cards such as the Bay Area’s Clipper card [9]; contactless payments via credit cards such as the Barclaycard [12]; and NFC-enabled advertisements in magazines [11]. The technology has been officially endorsed by technology giant Google [46]. NFC is the technology behind the Google Wallet payment application.

NFC is not without its detractors, however; some researchers have identified potentially vulnerabilities linked with the technology. For instance, researchers have demonstrated the potential for NFC-based misuse of real-world ticketing systems [27] and attacks against Google Wallet that leverage NFC [37]. Part of the trouble lies in the difficulty of testing NFC implementations prior to release, as discussed in [25].

In this project, we examine the automation of NFC transactions and then analyze two potential use cases for automated transactions. First, we consider the potential usage of NFC as an attack vector. More specifically, we consider NFC’s potential for performing denial of service attacks. Second, we consider the usage of automated NFC messaging for fuzz testing NFC. The project describes the implementation of a framework that performs automated sending of NFC messages that can be configured

to perform either fuzz testing or a denial of service attack against another mobile phone. We present an analysis of the impact of NFC-based denial of service on a mobile phone, considering aspects such as rate of battery exhaustion, reduction in network capacity, and reduction of available storage space on the phone. We will also explore some basic fuzz testing using the same application.

The format of this report is as follows: in Chapter 2, we present the necessary background information and related work on NFC, denial of service in mobile devices, and fuzz testing in mobile devices. In Chapter 3, we consider the motivations for exploring denial of service and fuzz testing in NFC. Then in Chapter 4, we present a basic mechanism for automating NFC transactions. Next we look at two specific use cases for the automation mechanism: Chapter 5 looks at using automation for performing denial of service and Chapter 6 looks at using automation for fuzz testing. Each of these Chapters presents some experiments to test the technology as well as a discussion of its strengths and weaknesses. In Chapter 7, we present some suggestions for future work. Finally, in Chapter 8, we provide conclusions and insights gained from the project.

CHAPTER 2

Background

2.1 Near Field Communication

Near Field Communication (NFC) is a low power, short range wireless technology used to trigger automatic transactions between two pieces of NFC-enabled hardware when they come into proximity. It has a maximum operating range of around 10cm, though in practice the transmission range is limited to 4 cm or less [19]. Its limited range is due to the fact that it works via induction. Every NFC device contains a small induction coil. When a powered device comes into range of another device, it generates a magnetic field that powers the dormant NFC circuit.

There are two general types of NFC devices relevant for understanding this project: mobile devices and tags. NFC mobile devices are generally cellular phones or tablets. Mobile devices can interact with other mobile devices or with passive tags. Tags are simply passive tags, very similar in nature to RFID tags. They cannot participate in peer-to-peer transactions but can be read or written by mobile devices.

In each NFC transaction, one party acts as the target and one as the initiator [13]. The terminology stems from the original NFC mode, with one NFC reader and a passive tag. In that case, the initiator refers to the device, which generates an RF field that powers on the passive tag [20]. The tag is then referred to as the target, as the initiator reads data from it. This vocabulary persists even in peer-to-peer NFC mode, where both devices are independently powered. The initiator is the device that will receive data, and the target is the device that will send data. To minimize confusion, we will refer to the initiator as the *destination phone* and the target as the

source phone throughout the rest of this work.

In most cases, the data transmitted via NFC is fed into an application, triggering some response. One common linkage is between NFC messages containing URI data and a cell phone’s web browser. When the mobile phone uses NFC to read a tag containing a URI, the web browser on the phone may be launched and directed to the specified URI. This kind of automatic transaction is at the core of NFC’s intended usage.

2.1.1 NDEF Format

Since NFC began as a technology for reading and writing passive tags, its message format was developed with tags in mind. There are several record types in existence, but the standard format is called NFC Data Exchange Format, or NDEF. A single passive tag can hold one NDEF message, and a peer-to-peer communication consists of passing one NDEF message (though the communication involves additional control packets as well) [31]. The NDEF message is simply a container for one or more NDEF records, as shown in Figure 1. An NDEF record contains one specific piece of data, which often represents a request for a particular action. For instance, an NDEF can contain an URI, which can be interpreted as a request to open the web browser and load that page. Other things that an NDEF may contain include a file, a phone number, or a Bluetooth pairing record [44]. Figure 2 shows an example NFC transaction.

The NDEF record format is detailed extensively in [31]. We summarize the format here in Table 1.

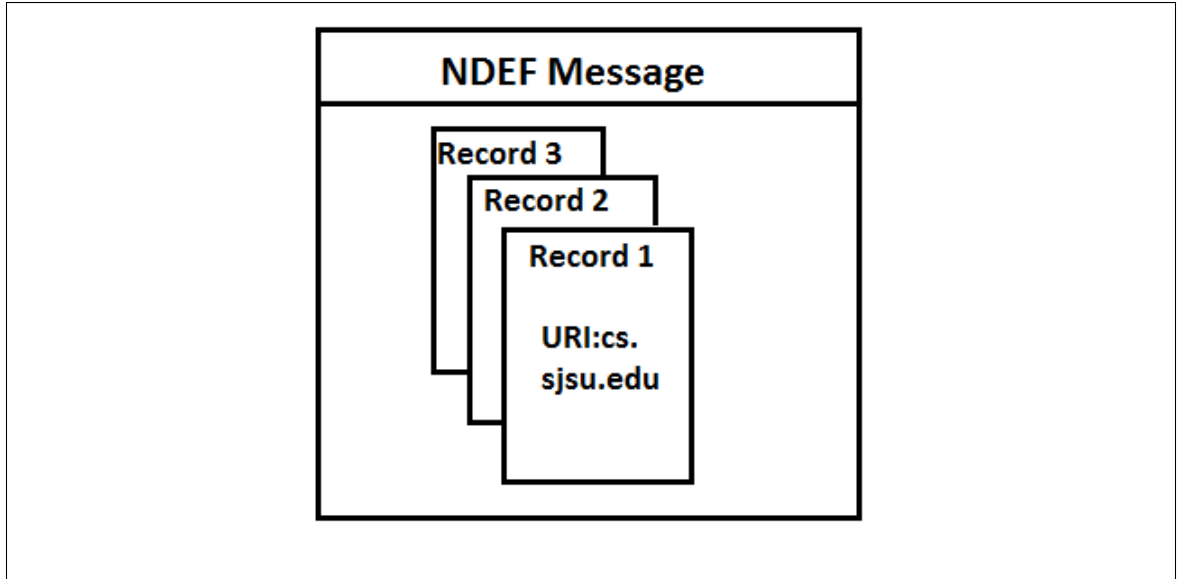


Figure 1: NDEF message containing multiple NDEF records

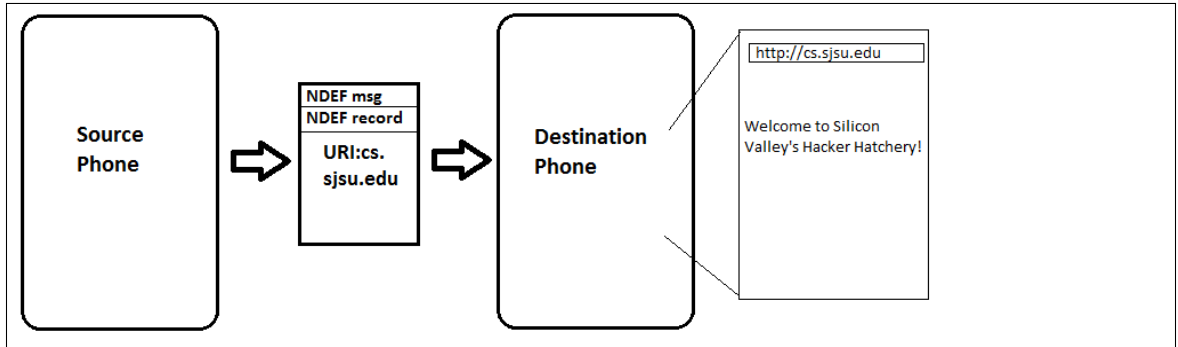


Figure 2: Example NFC transaction

2.2 Related Work

2.2.1 NFC Security

Many well-studied attacks on wireless communications, such as the man-in-the-middle attack, are impractical or impossible to implement in NFC due to its extremely short range. A good overview of such general attacks and why they do not work in NFC is available in [19].

Because more general attacks are not applicable, most of the NFC security research has focused on traits specific to the technology. One interesting issue unique to

Field	Length	Purpose	Notes
MB (Message Begin)	1 bit	Signifies first record in a message	Can be set for same message as ME
ME (Message End)	1 bit	Marks last record in a message	Can be set for same message as MB
CF (Chunk Flag)	1 bit	Chunk flag	Not used in Android
SR (Short Record)	1 bit	Payload length is only 1 octet long	Max payload length = 255 bytes
IL (ID Length)	1 bit	ID length field present	If IL=0, neither ID nor ID length included in record
TNF (Type Name Format)	3 bit	Specifies general category of payload	
Type Length	1 octet	Specifies length of type field in bytes	
Payload Length	4 octets	Specifies payload length in bytes	If SR = 1, payload length = 1 octet
ID Length	1 octet	Length of ID in bytes	If ID Length = 0, ID is excluded
Type	1 octet	Identifies content type	More specific than TNF
ID	1 octet	URI reference for the record	
Payload	variable	Record payload	May have a specific internal format

Table 1: NDEF record fields

NFC is that it is generally used as a pipe to carry a message directly to an application and trigger a response from that application. Thus, if NFC can be used to start and control an application (its intended purpose), then an attacker may be able to use NFC as a channel to exploit any vulnerabilities of that application.

One particular application of concern is the web browser. First considered by Mulliner in [27], the web browser presents an opening for attacks on on a mobile phone. Further research in this direction was presented several years later by Charlie Miller in [25]. Another application researchers have studied intensely is the previously mentioned Google Wallet. In [37], the authors the communication between the NFC adapter within a mobile phone and the secure element used for the Google Wallet application. Their primary finding is the potential for a software-based relay attack against Google Wallet.

Similarly, the interaction between the NFC service and other services present on mobile phones has caused concern for some researchers. In [44], Kooman and Verdult examined the relationship between NFC and Bluetooth in the Nokia 6212 Classic mobile phone. They discovered that NFC could be used to create a Bluetooth

connection, which in turn could be used to access and modify all of the memory on the phone remotely.

Another area of interest in NFC security research has been the NDEF message format and related vulnerabilities. Again, Mulliner was one of the first to the scene in this case, performing extensive message fuzzing to identify NDEF messages that would trigger unexpected behavior on the Nokia 6131 cell phone [27]. In a similar vein, Verdult and Kooman explored the NDEF message format in [44]. Their work focused on finding inherent flaws with the NDEF format rather than identifying particularly troublesome messages. They identified some grey areas in the NDEF specification that could lead to abuse in Nokia’s proprietary NFC implementation. Further attempts to perform and automate NDEF fuzzing in [25] also yielded a single bug in NDEF handling on Android that has since been fixed by Google.

One threat that has been proposed in several works but has not yet been implemented is the use of NFC as an instrument for performing a denial of service (DoS) attack. At least three published papers suggest the idea of implementing a DoS attack using NFC [27, 22, 42]. The authors of all three papers envision the attacker mounting the DoS attack using a specialized piece of hardware called a jammer rather than another NFC device. This is a likely result of the dates of publication. In 2009, NFC phones could not yet communicate in peer-to-peer mode and thus did not present threats to one another; however, modern phones have the peer-to-peer communication capability and thus now represent potential vulnerability. NFC has also been considered as a vector for performing denial of service against the mobile phone’s secure element via a replay attack [36].

2.2.2 Denial of Service in Mobile Devices

In brief, a denial of service attack is any attack that seeks to prevent the appropriate usage of any given resource by a user who has access rights to that resource [18]. Denial of service is a problem in many areas of computing; any time there is a limited resource, there is a potential for a malicious program to monopolize access that resource.

The inherently resource-limited nature of mobile devices makes them particularly vulnerable targets for denial of service attacks. In a mobile device, virtually all resources are limited, including: network capacity, battery power, memory, storage space, and CPU cycles. Mobile devices even have limited resources in terms of display space. Any of these resources could be at the center of a denial of service attack against a mobile phone.

The problem of denial of service attacks on cell phones has been considered in existing literature. Possibly the most frequently discussed vector for denial of service against mobile phones is Bluetooth, another short range wireless technology. Bluetooth differs from NFC in that it has a much longer range and faster data rate [44]. One similarity between the two, however, is that both push phones towards automating certain tasks. Bluetooth is often used to automatically pair mobile phones with peripheral devices. Bluetooth has been extensively considered as a possible vector for denial of service attacks [7, 17, 33]. Some of the papers focus on Bluetooth's potential for tying up network resources via signal jamming [7, 33]. Along with signal jamming, there has been research into using Bluetooth denial service to cause battery exhaustion [17].

2.2.2.1 Battery Exhaustion Attacks

Battery exhaustion is a particularly relevant type of denial of service attacks in cellular phones. As any routine cell phone user will know, battery life is a limited and precious commodity in a phone. If a battery exhaustion attack occurs, the battery life will diminish rapidly, severely interrupting the usage of the device. Three general ways of performing battery exhaustion attacks on mobile phones are presented in [24]: attacks that use excessive network activity to drain the battery life; attacks that use repeated valid tasks to drain the battery life; and attacks that use malicious applications or malware to drain the battery. In the literature, the first method, battery exhaustion via network services, seems to be the most popular by far. In [26], the authors study the impact of battery exhaustion attacks using WiFi and Bluetooth as the attack vectors, and find that the impact is significantly detrimental. The authors in [34] present an interesting attack using the cellular data connection as the attack vector. The cellular connection is used to quietly send massive numbers of text messages. The constant network activity causes the rate of battery exhaustion to increase.

2.3 NFC on Android

In recent years, NFC has become strongly associated with Android phones. The first NFC-enabled Android phone, the Nexus S, appeared in late 2010. Since then, more vendors have added support for NFC on their Android phones and Google has continued to expand its development on NFC for Android. In an effort to enable developers to create NFC applications and thus drive the demand for NFC in the market, Google has provided a set of intuitive APIs for NFC. Most of the NFC-related APIs are contained in the *com.android.nfc* package, with the *com.android.nfc.tech* also pro-

viding APIs specific to passive tag access [50]. Google maintains documentation on the NFC APIs as well as introductory information on programming with them [4].

2.4 Fuzz Testing and NFC

2.4.1 Fuzz Testing

Fuzz testing (also known as fuzzing) is a type of software security testing. It consists of bombarding a program with input in an attempt to find conditions that will make the software behave unexpectedly. When designing software, programmers tend to focus intently on the positive use cases for their product. The main goal is to design software that *can* do what the customer *wants* it to do. Less attention is paid to defensive programming, or anticipating bad or perhaps just unexpected behavior on the part of the user. Fuzz testing lifts some of the burden off of programmers by automatically generating well-formed yet potentially problematic inputs to a system.

2.4.1.1 Input Generation

There are a wide variety of ways to generate the input used in fuzz testing, and much of the existing research has focused on this problem. Input for fuzz testing falls into two general categories: generation based input and mutation based input [40]. Generation based input is essentially generating random input that fits a specified format. The format is given, but the contents are random. Mutation based input takes a set of inputs, often from a real program run, and then morphs those inputs to create new inputs. The input generation program may not even need to know anything about the structure of the data; it just needs to modify it a bit. Both methods have advantages and disadvantages [16, 40]. Generation based input may test a more thorough set of possible inputs, since it could theoretically generate any

valid message; however, it may test far more messages than needed to find bugs, thus wasting a lot of time and resources. Mutation based input is simpler and may be particularly useful in finding user data entry flaws, but may explore a much smaller input space, potentially missing many interesting cases.

2.4.1.2 Fuzzing Tools

Many tools exist for fuzz testing various systems and network protocols. One of the most popular and far reaching tools at present is the Sulley framework [1]. Sulley aims to be the be all and end of network fuzzing frameworks, providing a very flexible input generation system meant to allow for testing of a wide variety of protocols. Sulley is meant to be used from start to finish, however; it does not provide a lot of options for generating the input and then storing it rather than using it right away for testing. Another tool used for fuzzing is the SPIKE fuzzer creation kit [8]. Unlike Sulley, SPIKE just provides a language for developing fuzz test input. Delivery of the inputs is left to the user.

In both Sulley and SPIKE, the tester defines a block consisting of primitives representing the contents of the message to be sent [1, 8]. Each field is then fuzzed for testing. In both cases, the field may be both mutated or generated based on the settings.

Protocol-specific fuzzing tools exist for other niche wireless technologies such as Bluetooth and RFID [23, 43]. There is no existing widely used tool for generating NFC-specific input, but it is possible to use tools such as Sulley and SPIKE to generate inputs.

2.4.2 NFC-Specific Fuzzing

In the general case, creating the fuzzing inputs is considered to be the difficult problem to solve. Delivery is usually a simple matter. Consider the case of sending fuzzing input from a client program connected via WiFi to a server program. Once the work of generating the input is done, testing becomes a simple matter of forming packets and sending them across the link. This task is likely to be trivial to automate.

In NFC, however, the delivery is just as complicated as the message generation itself. By design, peer-to-peer NFC is not conducive to easy fuzz testing. The initiator phone queues one record to be sent over NFC. It connects to the target phone and transmits the single message, then it falls idle. In order to send the next message, the phones must be pulled apart and brought back together again. Obviously, this act does not scale well. When testing a couple of messages, it might be okay; however, the sheer physical tedium of moving the phones back and forth makes experiments including tens of thousands of inputs impossible.

Yet delivery is not the *only* challenge in fuzzing for NFC. Message generation for NFC is very specialized as well; the message structure is specific to the technology. NFC fuzzing has to take into consideration the various valid formats for NFC (NDEF and its variants) as well as the specific NDEF record types and their valid contents. There are a large number of possible permutations for valid messages for NFC.

Several efforts have been made to attempt fuzzing on NFC despite the challenges. One approach is to use a separate piece of purpose-specific hardware for performing the NFC transaction. Mulliner appears to have been the first to attempt this kind of fuzzing, using a standalone RFID reader/writer for message delivery and his own Python library for input generation [28, 29]. His work focuses on emulating the

interaction between an NFC device and a passive tag rather than two phones. He has provided a series of Python scripts for generating different types of NDEF records in [29]. He used these scripts to generate a set of fuzz testing inputs, but much of the work had to be done manually. Miller also attempted the problem of automating NFC fuzzing, this time using a ProxMark3 for testing against several NFC-enabled phones [25]. For this work, the input was generated using a modified version of the Sulley fuzzing framework. Miller’s work provides good insight into what to test for; he argues that Java exceptions caused by NFC messages can largely be ignored, but that native code crashes are interesting [25]. But again, it relies on a piece of specialized and somewhat expensive hardware.

More recently, Stirparo proposed an NFC fuzzing framework that utilizes the Android emulator within the Android SDK [39]. The work focuses primarily upon input generation and suggests an NFC-specific input generation system similar to Mulliner’s Python library. It also proposes a novel way of delivering the fuzz testing input. In the project proposed, the NFC messages would actually be events simulated via the Android debug bridge, possibly sent to an emulated device as Intents for the NFC subsystem to handle. Emulation provides some advantages in that it totally negates hardware costs and allows for testing a wide range of devices, and ADB is conducive to easy logging and monitoring; however, there are limitations to fuzz testing with the Android emulator. The use of an emulator may mask certain oddities such as synchronization or timing quirks, of which NFC has many.

Wiedermann presented an interesting project on fuzz testing for NFC in 2013 [47]. In Wiedermann’s design, the tests are generated using the Sulley framework (as in Miller’s work), but delivered to the phone via the Robotium tool. One advantage of Wiedermann’s design is that it requires little knowledge on the part of the tester

to set up. It also does not require specialized hardware, as the test runs between phones. The source phone must be attached to a computer running the Eclipse SDK, however; this limits the solution somewhat.

CHAPTER 3

Motivation for Automation

Before delving into the details of automating multiple message sending in NFC, it is worth taking time to consider how and why automation could be useful.

Historically, it made sense for NFC devices to only exchange a single NDEF message (or other tag) at a time. But the NFC usage paradigm may change now that NFC is available on more and more devices. There are many potential peer-to-peer mode applications where automation could be extremely useful. One current use case of Android Beam, Google's NFC file sharing API, is to transfer small files from one phone to another, such as applications or contact files [4]. With the need to physically decouple devices, Android Beam loses its utility quickly for non-trivial tasks. It's easy for users to exchange single contact files, but if a user wants to share his entire address book with a friend, are they to stand around bumping phones all day? If multiple message sending without physical decoupling were enabled, NFC could even have a purpose as an easy way of moving files around between one's own devices.

Another related reason to explore automating multiple message sending in NFC is that it would present a solution to some of the problems inherent in testing and securing NFC. As discussed in the previous Chapter, input testing is of immense value in developing secure software. It eliminates the need for the programmer to think of every possible disastrous mistake himself. But as it stands, NFC is very difficult to perform input testing on. The physical impossibility of testing anything near the set of possible inputs for even a simple program renders manual NFC fuzz testing pointless.

Automating multiple message sending removes the need to physically decouple the devices, instantly removing one of the major roadblocks in NFC input testing.

It is also worth exploring automating multiple message sending in NFC specifically because it does not follow the *expected* usage of the technology. One of the most wonderful features of the Android operating system is that one can download the source, tinker with it, and rebuild it oneself; however, the very openness of the operating system makes it extremely appealing as an attack platform. Attackers need not write their own OS or use an OS separate from that of their victims. With minimal Java programming skills, an attacker could easily replicate the system design presented in this report. So it is vital that researchers begin to consider these problems as well, before they become established attack vectors.

3.1 Comparison to Existing Solutions

In the previous Chapter, we discussed some of the existing solutions for fuzz testing (and thus ostensibly automating) NFC input. The existing research, while limited, has provided useful results already. Yet, it is worth continuing to pursue another solution. The main reason for this is financial. The existing solutions primarily focus on using standalone hardware, such as a dedicated RFID read-writer, to perform the message sending [25, 28]. The cost associated with such hardware may be detrimental to some researchers. There is also a learning curve associated with using unfamiliar hardware. In contrast, older generation NFC phones are now readily available for resale at reasonable prices, and most researchers will already be familiar with using an Android phone. So it is clear that performing automation via software on an Android phone has merit as an idea.

Another limitation of existing work is that it primarily looked at exploring phone-

tag interactions and not phone-phone (peer-to-peer in NFC parlance) interactions. As more phones appear with NFC capabilities, NFC peer-to-peer mode is likely to become the more prevalent paradigm. Thus any solution for testing NFC inputs should really target peer-to-peer mode.

Automation in software on physical devices is appealing for the reasons listed above, but one might wonder if it offers any advantages over automating in software using an emulator as some recent research suggests [39]. When going for the least expensive tested, using the emulator seems like an obvious choice; however, the emulator limits testing of traits inherent in the physical phone. The emulator cannot accurately measure the battery drain from using SNEP handover to transfer a number of files, for instance. The emulator may be suitable in some, but not all, cases.

Yet another potential method of performing automation is by using an automation tool such as Robotium [35, 47]. In the case of Robotium, the test is launched and monitored via a computer running the Eclipse SDK. That is to say, it is not possible to automate the task from a phone directly. The phone must be tethered to a computer. While having the phone tethered offers some advantages, such as easy logging and monitoring for results, it also limits some of the potential use cases for automation. Specifically, a denial of service attack would be entirely impossible if the device had to be connected to a computer.

3.2 The Relationship Between Denial of Service and Fuzz Testing

Recall that the aim of a denial of service attack is to prevent access to a resource by that resource's intended user [18]. One of the most obvious ways to accomplish this task is to consume the resource before the intended user can get to it. When the limited resource is something that can be affected by network traffic, then an

obvious attack method is to flood the network with packets. Thus, any practical implementation of a flooding attack is going to involve sending messages over and over again, likely without any significant pauses and certainly without any physical effort on the part of the attacker.

Now consider the act of fuzz testing. Fuzz testing is sending input to a system in order to reveal unexpected behaviors. Occasionally, manual fuzz testing may be feasible and appealing. There may be a couple of inputs that the programmer has likely not thought to handle correctly. (This is the basis of a great deal of browser-based hacking.) But in most cases, if a system allows for any sizeable set of possible inputs, manually inputting them one by one is unfeasible. It makes much more sense to generate a set of test inputs and then systematically send them to the system one by one, without any significant pause and again, without any physical effort on the part of the tester.

On the surface, the two tasks may seem very different. Figure 3 sums up some of the key points of each. Denial of service is a markedly “evil” sounding goal while fuzz testing seems to be “good”. In reality, they are both built around the same idea: automated message sending. Neither can occur on any significant scale without a means of transmitting an unlimited number of messages from a source device to a destination device automatically. Thus for either to occur on NFC, an automation mechanism such as the one proposed in this paper is necessary.

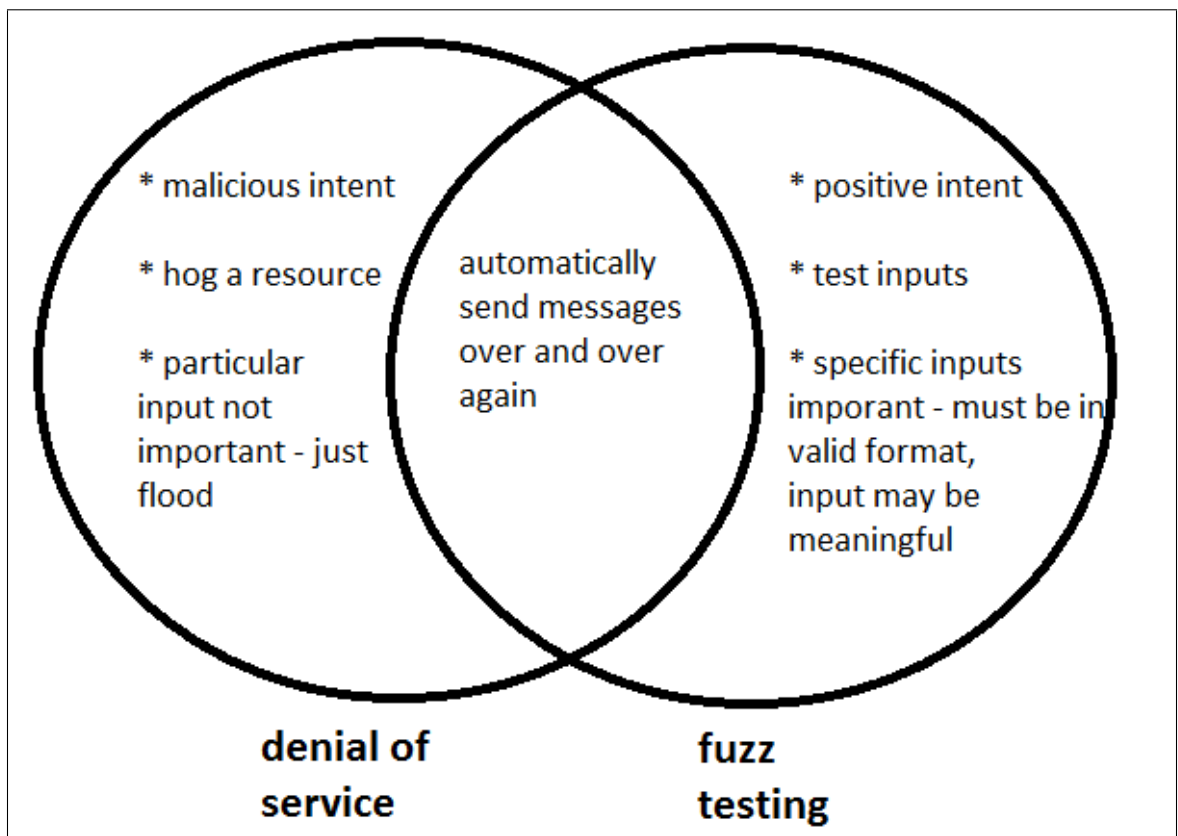


Figure 3: Comparison of denial of service and fuzz testing

CHAPTER 4

Automating NFC messaging

This chapter describes an ideal automation framework for NFC, where multiple messages can be sent without any need for human interaction. After presenting the ideal application, we then detail the modifications necessary to realize that vision.

4.1 An Ideal Automated NFC Application

Let us first consider our ideal automated NFC messaging application. The user should be able to specify inputs to the application. These inputs will form the contents of the NFC messages sent during communication. Some possible types of inputs to the application include a user-provided set of NFC messages, a randomly generated set of NFC messages, or a single hard-coded NFC message. The first type of input would work well for generation-based fuzz testing; the second type would work well for mutation-based fuzz testing; and the third type would be adequate for performing a flooding-based denial of service attack.

Once the input is provided, there should be a mechanism for beginning the automated sending. Once the mechanism is triggered, the provided messages should be sent, one by one, until some completion condition is met. Possible completion conditions include: end of specified test duration (for timed test), end of input file (for fuzz testing with a specific set of inputs), or the appearance of a failure condition (such as the NFC subsystem crashing).

It is critical that once the automated sending begins, there need not be any further human interaction with the device. That is, the attack or test should continue

until it hits a completion code, without a human having to mess with it. It should not be necessary to move the phones around, press any buttons, or manually interact with the device in any other way.

Ideally, the automated sending application would record some relevant statistics for the attack or test. It ought to record relevant statistics, including the duration of the test, the reason the test stopped, the number of messages sent, and details on the content of the messages sent. The duration of the test and the reason for completion give an insight into whether or not the run was successful. The number of messages sent may be useful for determining the minimum impactful attack, and as a confirmation that an entire data set was used for fuzz testing. The content source details would provide a reminder of the nature of the application run.

As an aside, it would be extremely nice to have statistics gathered on the destination side as well. Particularly in terms of fuzz testing, the results on the recipient's end may provide useful insight into what is happening. In our system design, we assume that the relevant messages will be captured by the Android logging system and available via the logcat tool [5]. Some other tools such as Sulley provide reporting on the target device, however, which is appealing but outside of the scope of this project.

Finally, any automated NFC messaging application should have at least a minimal ability to perform self-recovery. NFC has intermittent failures even in typical usage. A single error such as a debounce should not cause the entire system to grind to a halt. NFC is also prone to timing and synchronization issues, and these should not stop the application in its tracks.

4.2 Automated NFC Application Design

Our application is designed to run on an Android phone. It was specifically tested on a Galaxy Nexus HSPA+ phone. The solution designed includes two key parts: a modified build of the Android Open Source Platform (AOSP) operating system, and an Android application for configuring and launching the automated message sending. The design of the Section 4.4 will explain the necessity of building AOSP separately. A third important practicality is that the source phone must be rooted; Section 4.4.4 discusses this further.

4.3 The Application

The application was designed with the considerations outlined in Section 4.1. It consists of a simple UI that will meet the needs of both the fuzz testing and denial of service use cases, and a backend that performs the actual work.

4.3.1 Interface Design

The interface has a simple, straightforward design that will allow the user to configure either a fuzz test or a denial of service attack. Figure 4 shows the visual layout. It can be subdivided into two separate panes: the content configuration pane and the test length configuration pane. (Henceforth, "test" is used to refer to an execution of the message sending application, regardless of whether it is a fuzz test or a denial of service attack.)

The content configuration pane allows for the selection of the content source. The options are summarized in Table 2. Note that the selection of the content source determines whose responsibility it is to generate the message content. For options 1 and 3, the content is assumed to be generated by the user. For option 2, the content

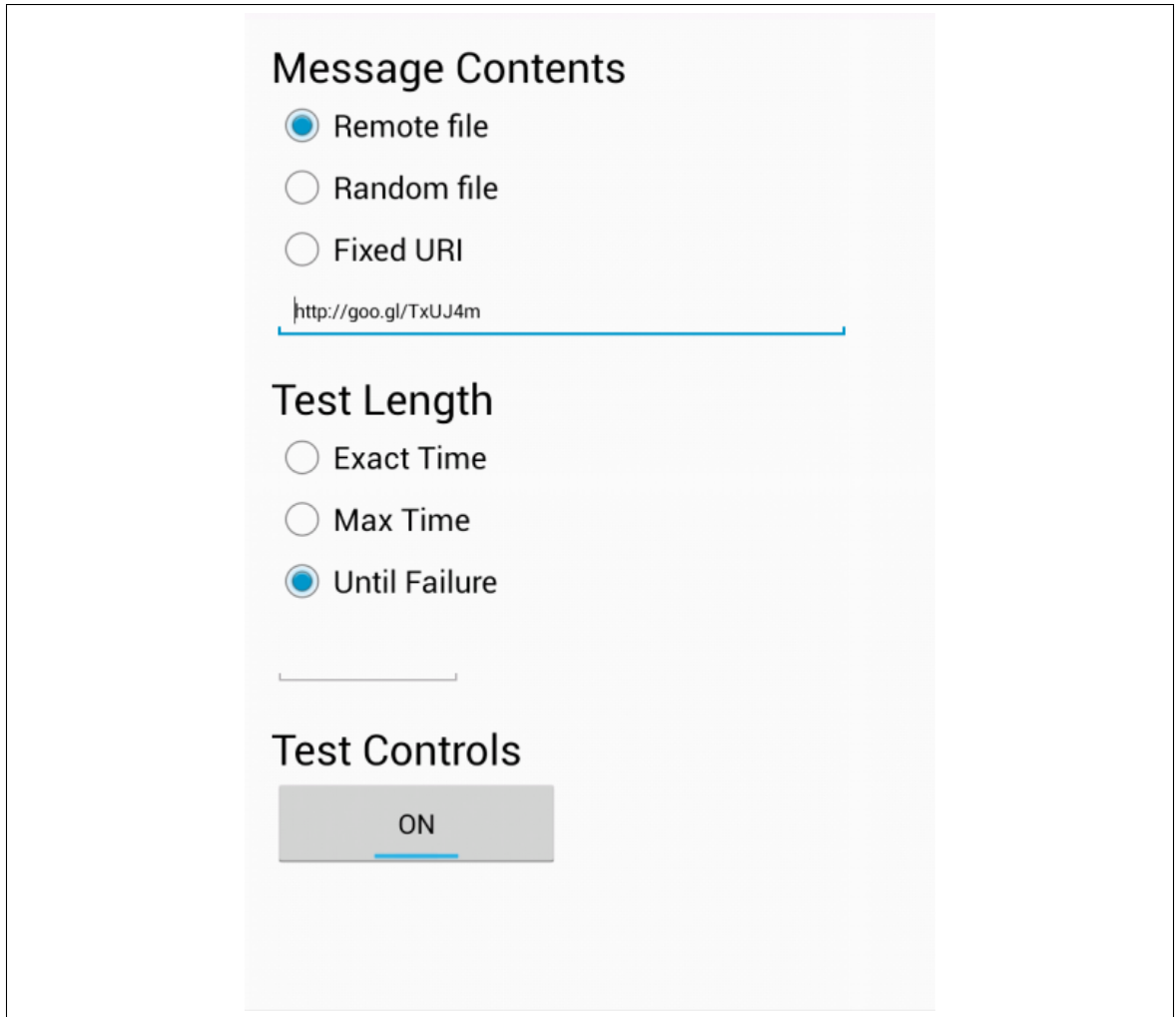


Figure 4: Automated NFC application user interface

is generated by the application. Section 6.4 discusses the method applied to generate these random contents for this project.

The test length pane allows for the selection of the completion condition for the test. The options are summarized in Table 3. The options are straightforward but it is worth mentioning the behavior that happens when there is a mismatch between the specified length and the content source; that is, what happens when the application runs out of input messages before the completion condition for the test is hit. When this happens, the solution is simple: we simply return to the beginning of the content

source and reuse the messages that were already sent from the beginning.

The third option, run until failure, requires a little more explanation than the other two. There are cases in which it is desirable to continue testing until an unrecoverable failure is hit. An unrecoverable failure in this context means something that irrevocably breaks the connection between the source device and target device. In practice, the two events that tend to constitute unrecoverable failures are NFC subsystem crashes and phone shutdown due to drained battery. NFC subsystem crashes occur frequently, but are often recoverable; indeed, the NFC service will attempt to automatically recover from subsystem crashes. Yet occasionally the subsystem will deadlock and crash somewhat dramatically. When this happens, NFC is completely unavailable until the phone has been restarted. Unfortunately, this can happen on either the source phone *or* the target phone, often during seemingly typical use. Phone shutdown due to battery drain can also happen to either party as well. Battery drain is likely to be a goal in a denial of service attack.

Along with the two primary panes, there is also a button for starting and stopping the test. When clicked from the stopped state, it does some basic checking to make sure the combination of content source and test length is valid, takes a timestamp, resets relevant variables, and launches the test. If it is clicked during the execution of a test, the test automatically stops and certain statistics are gathered. The reason for the test ending (user interaction) is also logged. In the event that the test completion condition is met without anyone manually clicking the stop button, the button will be automatically reverted to the off (click to start) state.

Note that during the execution of a test, all of the interface becomes non-functional aside from the stop button. This is to limit unexpected interactions between the UI elements and the Android event system during the execution of the test.

Option	Definition	Content Generation
Remote File	Messages are stored in a remote text file; the user will provide the URL in the provided field.	User-generated
Random File	Messages will be randomly generated using an NDEF fuzzing library.	App-generated
Fixed URI	The same message will be sent repeatedly; the user will specify a URI to be used as the message contents.	User-generated

Table 2: Content source options

Option	Definition	Content considerations
Exact Time	The test will run until the amount of time specified in the time field has elapsed, or an unrecoverable failure occurs.	If the content source is exhausted prior to the finish time of the test, the file will be reused from the start.
Max Time	The test will run until the time specified in the field has elapsed, or the content source is exhausted, or an unrecoverable failure occurs.	When the last line of the content file is hit, the test ends.
Until Failure	The test will repeat until there is an unrecoverable failure on either the source or target phone.	If using a content file, messages will be repeated as necessary.

Table 3: Test length options

When execution finishes, the UI elements will become available once more.

4.3.2 Backend

Now let us turn our attention to the backend of the application, or where the work is done. In this Section, we will describe the work done on the backend. The following Section (Section 4.4) will describe some of the work that needs to be done in order for the backend to achieve functionality. For now, assume that it is possible.

The idea behind the backend implementation is incredibly simple, though implementation is not entirely straightforward. The approach is this: after an NDEF message is sent, the next NDEF is queued for sending and the NFC adapter is *programmatically* reset the same way that it is reset when the phones are physically separated and brought back together. This action is repeated over and over again until the test's completion condition is met.

After an NDEF is sent, the next NDEF is pulled from the content source. (In the case of a fixed URI, the same NDEF is used over and over again.) Once the value is pulled from the input file, it is fed into our custom *createNdefmessage* function, which returns a new NDEF for sending. After the new NDEF is created, it is queued for sending using the *setNdefPushMessage* function (see Section 4.3.2.1). Only one NDEF may be queued at a time.

The key to repeatedly performing NFC transactions is to re-enter a state of *discovery*. Discovery is NFC parlance for the state in which one device is actively seeking out (attempting to “discover”) other tags or devices. In standard usage, a device re-enters discovery mode after it is found to be out of range of a device it had been communicating with. Our backend forces the phone into rediscovery by utilizing an API provided by the *NfcAdapter* class called *setP2pModes* (see 4.3.2.2). This is *not* a public API and must be revealed via reflection (see 4.4.5).

One quirk of the NFC subsystem is that it is very sensitive to timing. If the source device is forced into rediscovery at an inopportune moment, it is possible that it will miss its chance to become the target and instead become the initiator. Since the destination device is essentially unaware of the presence of the source device (it is simply receiving and never intending to send anything back), this can cause the entire test case to hang and eventually fail. The solution to this is to periodically

force rediscovery, *regardless of the state that the device is in*. That is to say, force the discovery cycle to begin again even if the device is already *in* discovery. Experimentally, 15 seconds is an adequate timeout for resetting rediscovery; almost all non-file NDEFs will have transferred in that time, so if rediscovery has not occurred, it is most likely due to a hang. Section 4.4.7 discusses the implementation details of this hard reset further.

4.3.2.1 Basic NFC Functions

In this Section, we review four NFC functions implemented and used by the backend. Figure 4 provides a summary of the functions discussed. Two of the four functions must be implemented by an Android application that wishes to use NFC: *onNdefPushComplete* and *createNdefMessage*. *onNdefPushComplete* gives the programmer an opportunity to enter a function to occur directly after an NDEF is successfully sent ("pushed") by the application. In our case, *onNdefPushComplete* is responsible for fetching the next input, using it to generate an NDEF, and calling the rediscovery function.

The next relevant function is *createNdefMessage*. In our implementation, the behavior of *createNdefMessage* is thus: the next line of input has already been successfully retrieved from the content source. (Note that if the content source is a fixed URI, we will only hit *createNdefMessage* once; after that, the same NDEF will be reused indefinitely.) The first character of input is scanned to ascertain the content type. There are two possible content types: simple URI or raw NDEF message.

Simple URIs are marked with the identifier *uri:*. If a line is detected as a simple URI, then *NdefRecord.createURI()* is used to generate an NDEF record containing the URI. This record is then used to create an *NdefMessage* object.

If the string does *not* start with *uri:*, then the input line is assumed to be an NDEF message expressed as raw bytes. The bytes are fed directly into a constructor provided by the *NdefMessage* class. We rely on the validity of the byte array in the line; if an invalid input has been provided, an exception may be raised. (See 6.4 for more information on how the input strings are built.)

SetOnNdefPushCompleteCallback simply marks a callback for the program. Since NFC is asynchronous, it must know where to return to after an NDEF push completes. Registering the callback by calling this function marks such a spot.

Finally, we consider *setNdefPushMessage*. This API has fallen out of favor due to the fact that it assumes static content; that is, the device will send out the same message to any other device it meets. A more modern API is *setNdefPushMessageCallback*, which allows the NDEF message to be edited on-the-fly depending on the encountered device. For instance, perhaps a plaintext message that simply said "hello" might be modified to say "hello Hal" if the device encountered a device whose name it knew. As we do not need this sort of dynamic functionality for performing either input testing or denial of service, we have opted to use the older, simpler API.

4.3.2.2 NFC Discovery, and Rediscovery

As previously mentioned, one of the core mechanisms in the application's backend is the NFC discovery loop. Discovery is the state in which the NFC device is actively scanning for the presence of other devices or tags. Discovery is controlled by the hardware abstraction layer (HAL), which iterates through a discovery wheel containing the possible transaction modes for the device. The cycling happens so quickly as to appear instantaneous, though in fact it sometimes results in timing issues. The possible roles for a device are active target, active initiator, passive target,

Function	Class	Purpose
onNdefPushComplete(NfcEvent)	–	Programmer-specified routine to occur after an NDEF is sent
createNdefMessage(NfcEvent)	–	Programmer-specified routine for creating NDEFs
createURI(String uri)	<i>NdefRecord</i>	Create an NdefRecord containing the URI specified by uri
NdefMessage(byte[] data)	<i>NdefMessage</i>	Create an NdefMessage object based on the raw bytes in data
setOnNdefPushCompleteCallback()	<i>NfcAdapter</i>	Provides a point for the program to return to upon NDEF push completion
setNdefPushMessage(NdefMessage, Activity)	<i>NfcAdapter</i>	Queues the next NDEF for sending

Table 4: Relevant NFCfunctions

and passive initiator [20]. The two active roles refer to NFCIP communication, where both devices generate their own RF fields. In passive mode, the initiator generates an RF field and the target modulates that field; this is meant for interactions between a device and either a tag or a device in card emulation mode. For our application, we wish the source phone to be an active target and thus it must encounter another phone acting as an active initiator (that is, scanning for other devices to read from).

In order to get into discovery, we rely on an API of the *NfcAdapter* class called *setP2pModes*. *setP2pModes* takes two arguments, *initiatorModes* and *targetModes*. These arguments specify whether or not the adapter is to play that role. In typical use, one of the arguments is 0 and one is 1; that is, the adapter seeks a particular role and not the other. This seems to determine which opportunity will be sought *first* on rediscovery. Since we wish our application to be the target, we pass in *initiatorModes*

$= 0$ and $\text{targetModes} = 1$. The API of the *NfcAdapter* class interfaces with a function in the class *NfcService* (which is not directly accessible by an application). *NfcService* contains a *setP2pModes* function as well. This function interacts with a *deviceHost* object representing the phone. It sets the phone's initiator and target modes per the provided parameters, then disables NFC discovery on the phone, then re-enables discovery on the phone.

So assuming we can access the public class *NfcAdapter*, we can call the *setP2pModes* function, which will do exactly what we want: make the phone appear to have left the presence of the other device (disabling discovery) and re-entered it (enabling discovery).

4.4 Barriers Preventing Automation

Of course, there is no such thing as a free lunch. A number of obstacles must be navigated in order to successfully automate NFC message sending.

The road to being able to trigger NFC discovery consists of many small steps. First of all, our application must be able to change the NFC adapter's state. To do so, it must first get the `WRITE_SECURE_SETTINGS` permission and a few other trivial permissions (4.4.1). This permission allows an application to modify the NFC adapter's state; however, *only* system applications may gain this permission. So we must convince Android that our application is a system application, and convince Eclipse that it is okay for us to request this permission. The latter requires changing setting in the SDK (4.4.2). To do the former, we must install the application in a special directory 4.4.3. To do that, we need to have permission to write in the Android file system— which is by default read-only. To make it not read-only, we need to root the phone and gain super user access (4.4.4). Once we have completed all

of these steps, we must gain access to the *setP2pModes* function, which is a hidden API. The easiest way to do this is by using reflection (4.4.5). Finally, everything should work! Well, almost. We won't be able to send some NDEFs without user input due to Android's Touch to Beam interface; so we'll modify the Android source to disable that 4.4.6. Then everything will work, except for when it doesn't due to a timing mismatch. We deal with the occasional mismatch and attendant system hang by periodically forcing rediscovery even when we haven't successfully sent a message (4.4.7).

4.4.1 Getting Permissions

Several permissions are needed in order for our application to work. Table 5 summarizes them. All of the permissions can be requested via the application's manifest; however, not all of them will necessarily be granted. The tricky one is `WRITE_SECURE_SETTINGS`. This permission allows the application to modify many sensitive objects on the phone, including the NFC adapter.

Permission	Purpose	Notes
<code>android.permission.NFC</code>	Allows the application to send & receive via NFC	
<code>android.permission.WRITE_SECURE_SETTINGS</code>	Allows the application to change the state of the NFC Adapter	Only granted to system apps
<code>android.permission.INTERNET</code>	Allows the application to utilize WiFi directly	Needed for retrieving remote content source. NOT necessary for NFC handover to WiFi.
<code>android.permission.ACCESS_NETWORK_STATE</code>	Application can view information about network state	

Table 5: Necessary permissions for the automation app

As mentioned in 4.3.2.2, our scheme relies on being able to force the phone into

discovery mode programmatically. In order to do this, we must be able to change the adapter's state; however, this is disallowed in Android for fairly straightforward reasons. If one can change the NFC adapter's state directly, then one can force the adapter to be turned on without the user's knowledge. In doing so, the user may unknowingly be put at risk. (Even if he doesn't hit a security vulnerability in NFC, he will lose some battery life to the NFC subsystem.) Because of this, the most critical permission for our application, `WRITE_SECURE_SETTINGS`, is granted only to system application and is not intended to be available to third party apps such as ours.

The first roadblock with acquiring this permission is easy to get around. The Eclipse SDK will, by default, disallow inclusion of this permission the application's manifest. A fatal error will be thrown and the application will be shown in error. To remedy this, one simply needs to change the lint preferences in the Eclipse IDE. Details on this appear in 4.4.2.

The next roadblock, installing the application as a system app so that it can actually gain the permission, requires more work. This will be discussed in 4.4.3 and 4.4.4.

4.4.2 Pacifying the SDK

To convince Eclipse to allow the application to request the permission, we had to disable the Lint warning about system application permissions. To do this, one can select Project->Properties->Android Lint Properties from the Eclipse menu. The option ProtectedPermission should be selected and disabled. Figure 5 shows a screenshot of this process.

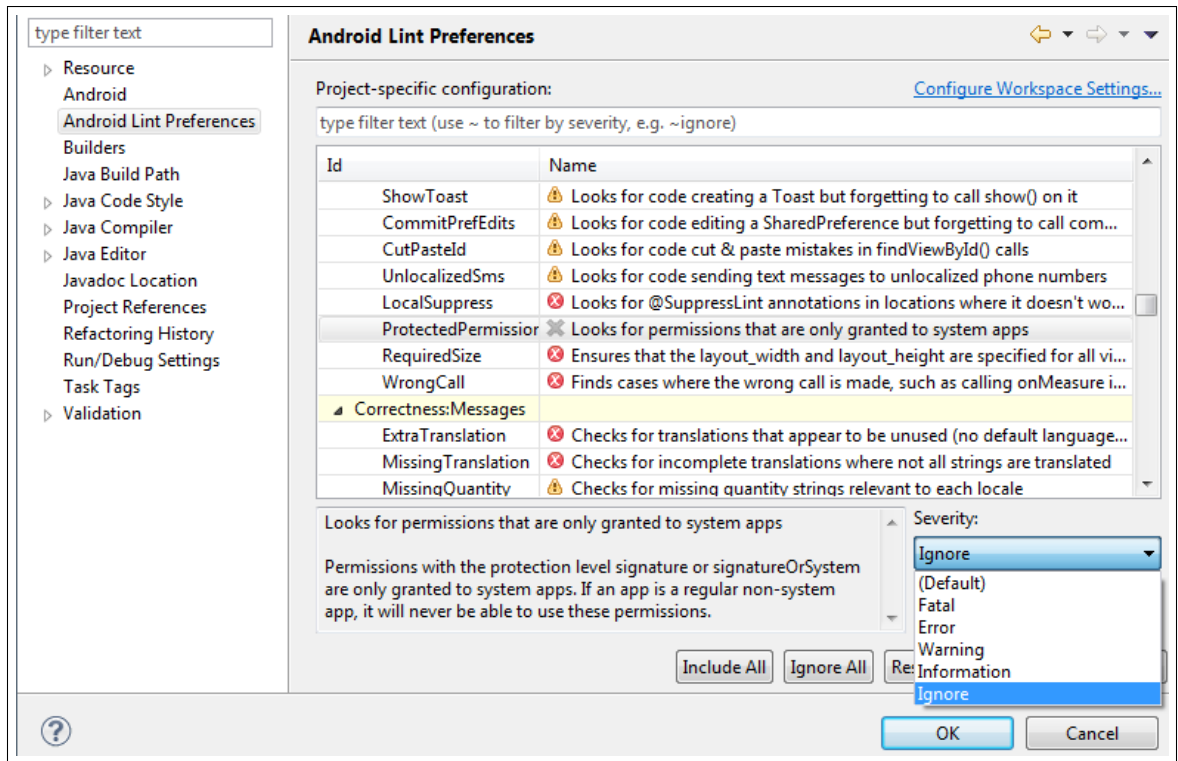


Figure 5: Disabling the Lint warning

Step	What is modified
Add permissions	Application manifest
Change lint preferences	Eclipse environment
Root phone	Phone environment
Install as system app	Phone environment
Disable touch to beam	Android source
Gain access to p2pModes via reflection	Application

Table 6: Solutions to problems

4.4.3 Installing as a System Application

Ordinarily, third party applications in Android are installed to the directory . Applications in this directory have limited permissions within the system. Operating system applications, ones written by Google and necessary for the functioning of the phone, are installed in and given special privileges. Elevating an application to

```

[root@localhost android-repo]# adb push wormui.apk /sdcard/
1010 KB/s (262324 bytes in 0.253s)
[root@localhost android-repo]# adb shell
root@maguro:/ # su
root@maguro:/ # mount -o remount,rw -t yaffs2 /dev/block/mtdblock3 /system
root@maguro:/ # cd /system/app
root@maguro:/system/app # cp /sdcard/wormui.apk .
root@maguro:/system/app # chmod 644 wormui.apk
root@maguro:/system/app # cd ../../
root@maguro:/ # mount -o remount,ro -t yaffs2 /dev/block/mtdblock3 /system
root@maguro:/ # reboot
[root@localhost android-repo]# █

```

Figure 6: Installing the application as a system application

“system application” status is merely a matter of installing the application in the correct directory.

Naturally, this is not a straightforward task. By default, is read-only. Applications cannot be copied to the directory either via the shell or by using adb push. The way to get around this is to remount the file system at as read-write rather than read-only. The overall process is push the application to the SD card, remount the file system, move the application from the SD card to , assign its permissions, remount the file system as read-only again and reboot the phone (which will trigger installation). Figure 6 shows the exact commands used for this process.

But there’s a catch. The filesystem cannot be remounted unless the user has sudo. That is, the normal user in the adb shell cannot use mount. To do this, one must have super user access, and to gain that, the phone must be rooted.

4.4.4 Unlocking and Rooting the Phone

Unlocking and rooting an Android phone are two closely related concepts. Unlocking refers to gaining full access to the bootloader. This lets us replace the stock installation of the Android operating system on a phone with a custom version of the operating system (often referred to as a custom ROM) [45]. This will be required to

disable the Touch to Beam UI as discussed in 4.4.6. Unlocking can be done through the *fastboot* tool or via an unlockingpackage.

Rooting is an additional step beyond unlocking. Android is built largely as a Linux distribution, and as such, many sensitive system tasks are restricted to users with root permissions. But unlike the average Linux distribution, stock Android does not provide any mechanism for assigning the owner of the phone root privileges. The exact process for rooting varies depending on the phone being rooted. The easiest way to root an Android phone is to find a tool for the desired phone and use it. For this project, the WugFresh Nexus Root Toolkit v1.8.2 was used to unlock and root the phone, as well as to perform periodic backups. Appendix A.2 discusses this tool and its use further.

4.4.5 Using Reflection to Access *setP2pModes*

At last, our application has the correct permissions to modify the NFC adapter state. Yet it still cannot get to *setP2pModes* directly, because *setP2pModes* is a hidden API. One way to get around this is to use reflection to gain access to the API. Reflection is a Java programming technique that allows for inspection of the metadata that exists for all classes at runtime [38]. This metadata includes a lot of very useful information, such as the names of all the functions and members of a class. If a class has private members, those are listed as well. If one knows (or can guess) the name of a member or a function, built-in Java reflection functions can be used to search the loaded classes for that member or function. After it is found, it can be set as accessible.

By reading through the Android source code, we were able to pinpoint the relevant class (*NfcAdapter*) and the desired API (*setP2pModes* as well as the relevant

parameters for the function. (The aforementioned *initiatorModes* and *targetModes*.) During runtime, our application calls a function that causes the classes of *NfcAdapter* to be enumerated. This list is examined to locate *setP2pModes* and a handle is retrieved for that function. After that, we are able to use the handle to call the function.

4.4.6 Disabling the Touch to Beam interface

The Touch to Beam UI is a recent addition to the Android platform. Touch to Beam specifically affects NDEFs carrying files, as the files will be transferred using the Android Beam API. The way it works is that the user queues up (with the help of an application) an NDEF containing a file to send to another phone. When the phones come in contact, a message appears on the sender's phone requesting that he tap the screen in order to continue the transaction. Unless the screen is tapped, the queued NDEF will not be sent and the transaction will simply hang, waiting for user input or for the phones to move out of range. Figure 7 shows the UI.

The Touch to Beam UI presents a problem if we wish to automate sending multiple NDEFs containing files, since our goal is to eliminate human interaction with the phone and Touch to Beam requires a tap per message. The fix for this is to simply disable the UI in the Android source and rebuild it, then flash the phone with our (very slightly) custom ROM. (It may seem like overkill to rebuild the entire source just to disable the UI; however, it also made it possible to add a massive amount of debug output to the NFC subsystem, which definitely aided in our understanding of it.)

In order to disable the Touch to Beam UI, the Android source must be very slightly modified. The source contains a class called *P2pLinkManager*. This class



Figure 7: Touch to Beam UI [41]

provides high level management of the link state for peer-to-peer NFC transactions. In NFC, Logical Link Control Protocol (LLCP) is the layer-2 protocol. *P2pLinkManager* monitors the state of LLCP and performs certain actions based on what is occurring at layer 2. When a new LLCP connection is activated, *P2pLinkManager* checks to see if a flag, `FLAG_NDEF_PUSH_NO_CONFIRM` is set. This flag gets set for NDEFs containing files, e.g. the variety pushed over Android Beam. When the flag is set, the function *onP2pSendConfirmationRequested* is called, which triggers the appearance of the Touch to Beam UI. We circumvent this by removing the call to *onP2pSendConfirmationRequested* regardless of the status of the flag; thus the behavior for a non-Beam NDEF is triggered instead.

Note that only the source phone needs to be flashed with the custom ROM. The destination phone should not be. In a denial of service attack, we want to be able to assume that the target phone is stock Android, and so do not wish to place a custom ROM on it. For the purposes of fuzz testing, even minor changes to the NFC stack may change the phone's behavior so again, it's best not to use a custom ROM. As a result of this, the destination phone may in fact see the touch to beam UI message

appear at certain times. It is assumed that there will be no human present on that end, and the message will be ignored.

4.4.7 Periodically Resetting NFC

The NFC subsystem is complicated and relies on many asynchronous tasks happening at once. Naturally, it sometimes suffers from synchronization and timing problems. Unfortunately, these may cause our automation application to hang. The solution to this is to set a timer and periodically restart NFC. We implement this using Java's built in *Timer* and *TimerTask* classes. *TimerTask* specifies a function that should occur when a timer goes off. *Timer* represents the timer itself, and has a method called *scheduleAtFixedRate*. *ScheduleAtFixedRate* fires the associated *TimerTask* at regular intervals. In our case, the *TimerTask* checks how long it's been since the last NDEF was queued. If it's over a certain threshold, then discovery is disabled and enabled again without changing the NDEF contents. It is extremely likely that the phones will synchronize correctly on the second try, and the automation will resume as before. (Note that if it does fail again, discovery will be disabled and re-enabled yet again the next time the *TimerTask* fires.)

4.5 Logging

One final topic that is shared between both use cases of our automation application is the logging mechanism. Whether automation is used for fuzz testing or a denial of service attack, it is useful to gather some statistics about the program's execution for later examination. In fuzz testing, the use is practical and ongoing: we want to see if anything unexpected happens on the source phone's part during the testing, and we want to be able to correlate what we see on the source phone side with

what we see on the destination phone side. In a denial of service attack, logging may not be required; however, in the *development* of a denial of service attack, logging is essential in helping us determine the impact of the attack.

The logging mechanism used by the application is extremely simple. The first time the automation application runs, it creates a directory on the phone’s SD card for storing the logs called *nfc_automation_logs*. The SD card is a particularly convenient place to store the logs because data stored on the SD card is accessible externally. The directory’s contents are accessible from tools such as adb, via Windows Explorer, or even from other applications. Thus it is very easy to retrieve the logs if they are stored on the SD card, and should make it easy if the logs are later fed into another tool for analysis. (See Chapter 7.)

Then, when a test is started, it opens a file in this directory, with a file name consisting of the word *log* plus the current time in milliseconds. For instance, a test started around 8:25 p.m. (GMT) on April 11, 2014 would have an automatically generated file name of *log1397247877297*. This naming scheme assures that the file names will not be repeated.

At the start of the test, the current time is recorded along with the test configuration parameters (content source and test length). Statistics are updated throughout the test. At the end of the test run, the gathered statistics are added to the file along with a final timestamp. The file is then closed. The values are written to the file in CSV format to make for easy parsing by other scripts. Table 7 shows the values logged for all runs of the application.

Value
Content source parameter
Test length parameter
Start time
End time
Total run time
of NDEF messages sent
of forced resets
Completion condition
Exceptions caught

Table 7: Values logged

CHAPTER 5

Using Automation for a Denial of Service Attack

In this Chapter, we consider the use of the automation application described in Chapter 4 for mounting a flooding-based denial of service attack via NFC. The basic idea behind the attack is to continually sending requests to the destination phone, referred to as the victim throughout this Chapter, until the victim reaches a failure state. Possible failure states include complete battery drain, in which case the attack may be classified as a battery exhaustion attack; or complete disk usage, in which case the attack may be classified as a storage consumption attack.

5.1 Entities

Both of the phones in question are Galaxy Nexus GSM+ phones with Android JellyBean 4.3 installed on them. One phone, previously referred to as the source phone, is the **attacker**. The other, previously referred to as the destination phone, is the **victim**.

The victim phone is considered to be innocent with regard to the attack. It is running stock Android with no modifications to the source code. It does not need to be unlocked or rooted for the attack to work. It does not need to have any special applications installed for handling NDEF messages, though the attack will continue even if the victim has custom NFC applications installed. In the absence of any non-standard applications for handling NDEF messages, the victim phone will resort to Android's default handling of received NDEFs. The victim is truly just the unfortunate recipient of the attacker's messages, and indeed, no human interaction with the victim phone is necessary to start or continue the attack.

The attacker instigates the attack and is responsible for sending all the messages. The attacker must be modified in the ways described in Chapter 4 to allow for the automation application to run. That is, the attacker phone has been unlocked, rooted, flashed with a custom ROM that removes the Touch to Beam API and allows for the collection of debug information, and had the automation application installed as a system application. The attacker phone is assumed to be manned at least initially; that is, a human must launch the attack.

5.2 Interactions

To launch the attack, the attacker uses the previously described automation application. He selects any content option and the test length option "run until failure" and then presses the start button. Note that the attack will not begin until the victim phone comes into range; nor will it start before the start button is pressed. So a real-world attacker could have the attack ready ahead of time, then simply bring the attacker phone into range of the victim phone at the chosen time. (This is perhaps more subtle than queuing an attack in real time.)

Once the attack has been launched, it will continue until a failure condition is met. Ideally, the failure is on the part of the victim phone, and consists of the victim phone running out of battery life, running out of storage space, experiencing a catastrophic crash of the NFC subsystem (at which point NFC is disabled until the phone is restarted), or meeting with some other unfortunate end. However, if the NFC subsystem has a hard crash on the attacker phone, that also counts as a failure condition and the attack will necessarily stopped. The final possible failure condition is a timeout. If it has been over 10 minutes since an NDEF was last sent successfully, the automation application will cancel the test.

The attack is expected to have a significant impact on battery life, particularly when WiFi handoff occurs. If the victim phone receives a file, either through a URI of a type that the Android browser immediately downloads or via an NDEF containing a file, the attack will also have an impact on storage space. It is also possible that if the same URI is sent to the victim over and over again, the victim phone could actually be flagged for abuse—since to a web server, the victim would seem to be the one performing a flooding attack. We will investigate the first two impacts later in this Chapter.

5.3 Attack Assumptions

The denial of service attack will work against stock Android Jellybean 4.3 with a few caveats. First, the victim’s screen must be unlocked and on. This is not an insurmountable barrier; it has been shown that it is possible to auto-unlock the screen of an Android phone given the victim’s phone number [25]. The victim’s screen must stay on for the duration of the attack. Again, this could be achieved using Miller’s phone number exploit, used repeatedly. Some NDEF record types may also cause the screen to acquire a wakelock, but this depends on the configuration of the victim device. If the application that is registered to handle a record type acquires a wakelock, then sending the message will keep the screen on. Android’s default NDEF handling does not seem to do this, though. So in practice, we achieve the screen conditions by installing a free application, Keep Screen On Free, to keep the victim phone’s screen turned on throughout the attack [21].

The victim’s phone must stay in range of the attacker’s phone for the attack to proceed; however, note that the attack does not automatically *fail* if the victim phone leaves proximity. It will eventually timeout if at least one NDEF has been sent

successfully since the start of the attack yet no NDEFs have been successfully sent in the past 10 minutes. In the meantime, the attacker phone will still advertise itself as a potential target, and if the victim phone returns to proximity, the attack will seamlessly resume.

Obviously, both phones are assumed to be NFC capable. The NFC adapter must be enabled on the victim phone for the attack to work. At this time, the automation application is not able to force the victim to enable his adapter.

The attack was only tested on Galaxy Nexus phones. It should ostensibly work on any other phones running Jellybean, but no such phones were available for testing at the time of this report.

5.3.1 How Close is Close Enough?

It is worth taking a moment to discuss what proximity means in the context of our attack. In the literature, various numbers appear for the acceptable range for NFC transmissions. Up to 4 centimeters is a commonly repeated guideline, yet this did not seem to line up with our results during experimentation. A simple experiment was performed to measure the actual range of operation between the two phones used for the rest of the experiments. The automation program was started in attack mode (as described in 5.2) and set to send a fixed URI for 5 minutes. It was then run for 5 minutes at each of 0.5cm distance intervals, starting at 0.0 and continuing to 4.0. During each 5 minute interval, we recorded statistics on the number of NDEFs successfully sent via NFC and the number of times NFC had to be manually reset by the application. Figure 8 shows the results of this experiment.

The results show that in typical usage, the successful transmission range is roughly 3.5cm, slightly less than the commonly touted number, and nowhere near

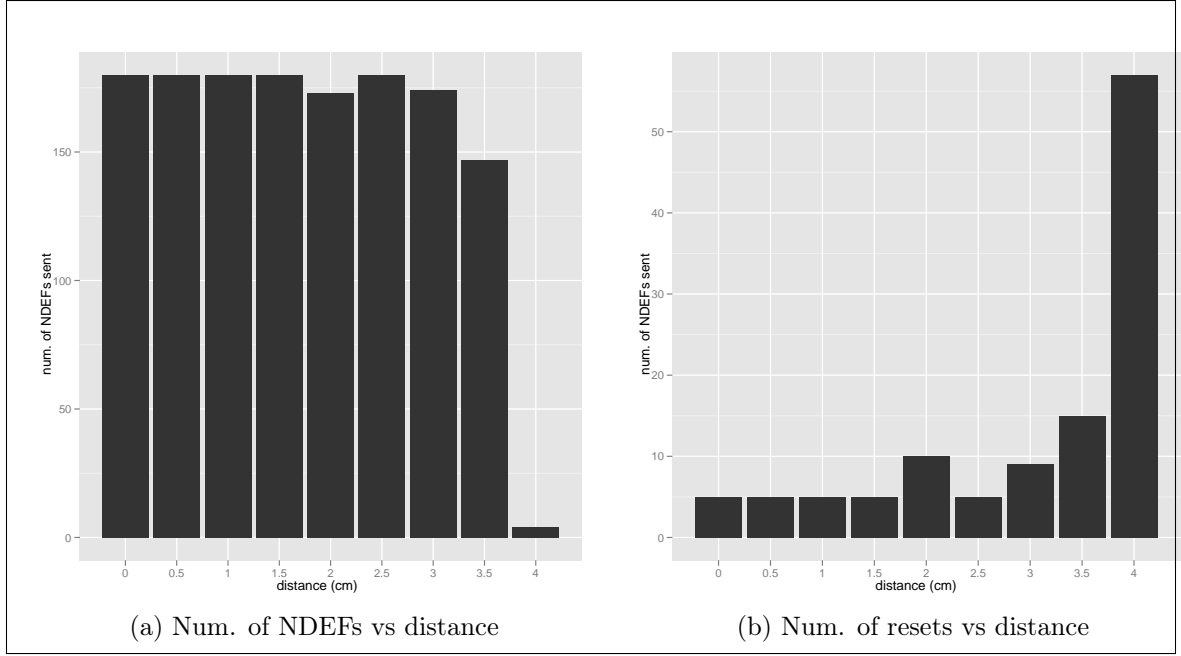


Figure 8: Performance of NFC at various distances

the theoretical maximum of 10cm.

5.4 Attack Implementation

The potential usage of our framework for a denial of service attack was assessed using the application described in the previous chapter.

The scenarios presented focus on monitoring the battery drain and storage consumption associated with a denial of service attack over NFC. For these experiments, we focused on sending NDEFs containing fixed URIs. The reason for selecting to send NDEFs containing fixed URIs is a matter of trying to maximize power consumption by the attack. By default, when an NDEF containing a URI is received on Android, the URI is opened in a web browser. This means that the NFC connection triggers a WiFi connection to fetch the URI. Outside of the phone's display (which is assumed to be on throughout the attack), WiFi is known to be one of the greatest power drains

on most mobile devices, so it is a good choice for investigating a maximal battery drain during denial of service [10].

Three scenarios were evaluated:

1. **Idle phone versus phone under attack:** In this first simple scenario, we monitor the phone while idle for 5 minute intervals. During this time, the victim phone runs only the Keep Screen On application, Power Tutor, and our victim monitoring application. Battery statistics are collected. We then compare these results to 5 minute intervals with the phone actively under attack. In this scenario, the attack consists of sending links pointing to a 50kb .apk file.

The impact on the victim phone’s battery was measured in several ways. For a coarse grain view of the battery drain from the denial of service attack, we used a small custom application that runs on the victim phone. The application writes a log file to the SD card on the victim phone. Every 30 seconds, this log is updated with a current timestamp and the current battery level. The battery level is retrieved using the Android BatteryManager class and its APIs.

For a more in-depth view of how the attack is impacting the battery, we use the application PowerTutor 1.4 [49]. PowerTutor also runs as an application on the victim phone and measures the overall power consumption of the system over some user-specified interval. It also shows the power consumption percentage of various applications over the interval as well as the power consumption percentages for various system components (CPU, WiFi, Display, etc).

2. **Content source: fixed URI (file download) + test length: 10 minutes, varying file size**

In this scenario, we looked at the impact of sending NDEF messages containing

URI records pointing to .apks (which are automatically downloaded by the Android browser) of varying sizes. We tested file sizes of 50kb, 500kb, and 5MB. The idea behind this scenario is that if larger files can be downloaded during an interval of time, the storage space will be consumed faster. We also looked at how varying the file size impacted the battery drain.

3. **Content source: fixed URI + test length: exact time, varying test length**

In this scenario, we tested a fixed URI as in scenario 1, but used a variable length fixed timed test. This allowed us to ascertain whether or not the power drain from NFC was consistent over time. The intent was to be able to suggest an achievable per-minute battery drain as a result of denial-of-service via NFC. We tested intervals of 10, 20, and 30 minutes and again monitored with our victim monitoring application and PowerTutor.

5.5 Results

In Figure 9, we present our results for the first scenario. Figure 9a shows the kernel densities of the amount of energy drained from the battery over the course of the experiment. The results for the idle phone are very consistent; we can see that almost all of the phones lost 3 percent battery life over the course of ten minutes. This may sound high at first, but in this scenario, we kept the phone’s screen on. The screen is the largest power drain on the phone, so even though the phone is almost entirely idle, it still loses a significant amount of battery.

The kernel densities for the phone under attack look similar in shape but are centered around 6 percent with a larger spread than the idle phone. Figure 9b shows the average for both scenarios. The active attack drains the battery at nearly twice

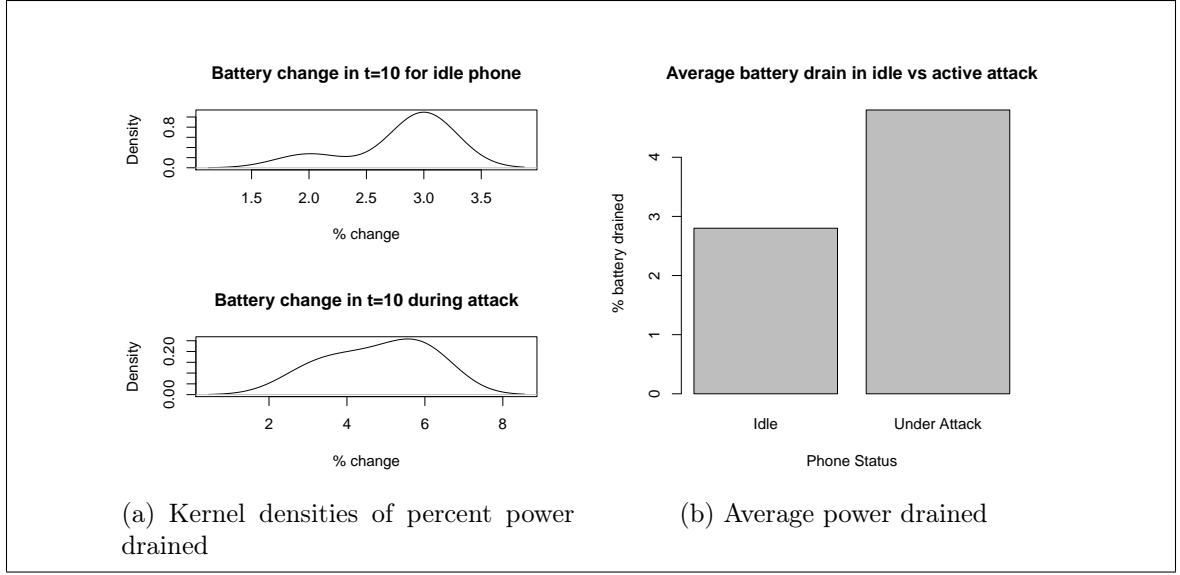


Figure 9: Idle vs active

the rate as in the idle scenario. The mean battery drain in the idle scenario is 2.8 percent over the course of ten minutes. The mean battery drain in the attack scenario is 4.8 percent over the course of ten minutes.

From this initial number, we can extrapolate that it would take approximately 3.5 hours on average to drain a full battery. Initial experiments indicate that this is correct; however, fully draining the battery without encountering an additional problem first is uncommon. We describe some of the many problems that appear on the victim phone as a result of our attack in A.5. It is also worth considering the fact that very few people tend to walk around with fully charged cell phones.

In Figure 10, we present two attempts to drain the full battery of the victim phone. In one case, the phone is idle save for Keep My Screen On and the victim monitoring application. In the other case, the attack continued until the victim phone's battery was completely drained. We can see that these results closely follow the results seen in the shorter tests. During the active attack, the battery drained

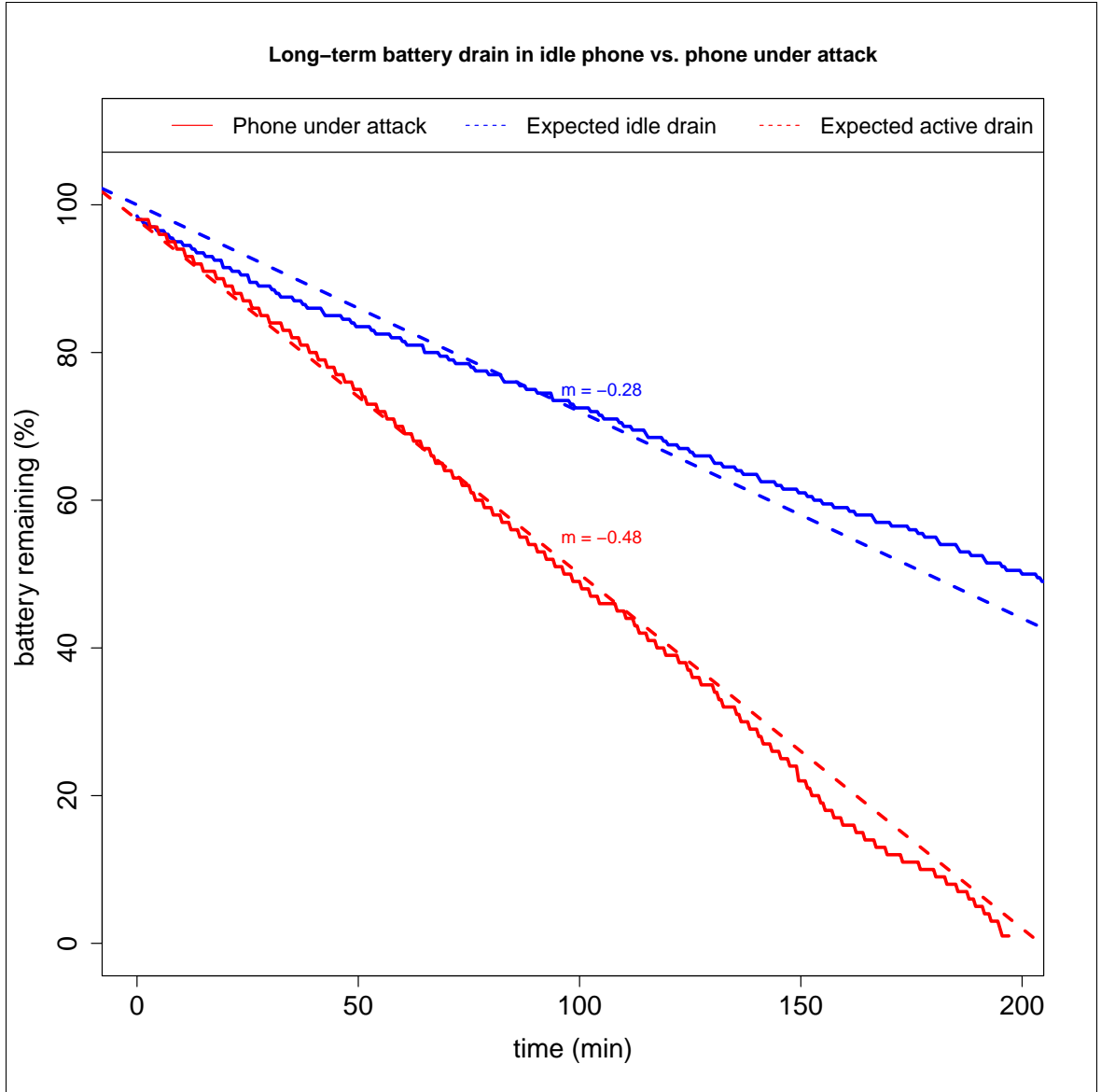


Figure 10: Battery drain over extended time period

slightly faster than predicted. In the idle scenario, we again see that the battery drain is slightly slower than predicted, but its slope closely follows our predicted rate.

5.5.1 Varying file size

Figure 11 shows how varying the size of the file linked in the NDEF impacts the victim phone's battery. Each subgraph shows the kernel densities for the battery

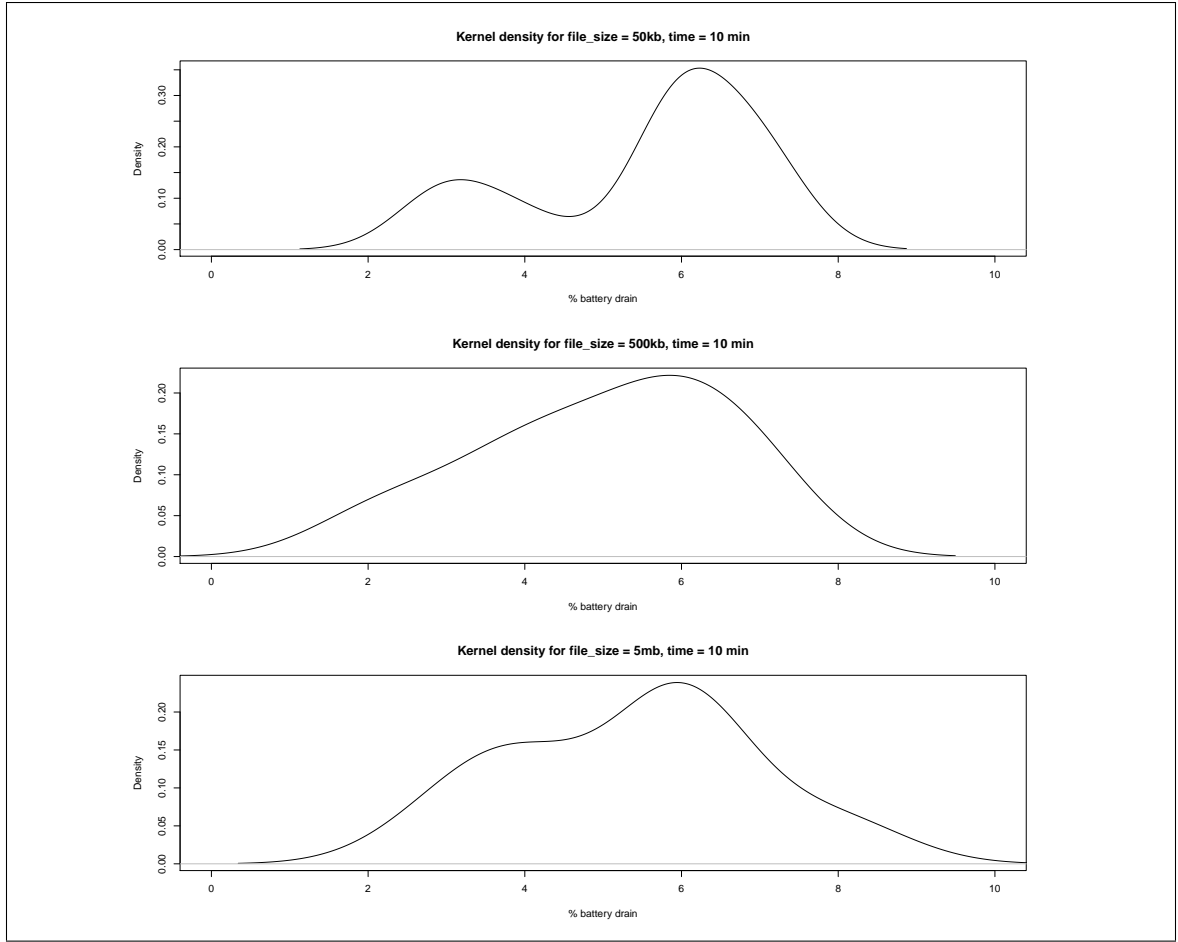


Figure 11: Battery drain for various file sizes

drain for that file size. For all three file sizes, the kernel densities peaked at 6 percent. Performing t-tests on the raw data indicated that there is little difference between the three groups in terms of battery consumption. Thus if battery exhaustion is the only aim of the attack, there is no advantage in using larger file sizes.

Figures 12,14,and 16 show the storage consumption recorded across ten trials for each file size. Figures 13,15 and 17 show the average values across all 10 trials for each file size group to illustrate the overall trend.

All three file sizes follow a similar pattern. We can see that the available space begins to decrease at the beginning of the attack, then increases temporarily before

resuming its downward trend. The jump upwards has to do with the Android browser cache, which is cleared once it hits a certain size. Overall we can see a downward trend across all three groups, with the largest removal of free space coming from the file size = 5Mb; however, we can also see that the 5Mb group is by far the least consistent.

During the course of the experiments, we found that the 5Mb file size lead to many problems on the victim phone. In the course of one 10 minute experiment, it was not uncommon for the Android browser to crash multiple times, for the NFC subsystem to crash, or for the phone to simply shut down. Appendix A.5 discusses these issues in somewhat more detail.

One additional aspect we considered in the file size experiments was the number of NDEFs sent by the attacker. Figure 18 shows that regardless of the linked file size, the attacker can send about 400 messages in 10 minutes. Thus, the attacker can kick off 400 file downloads of any size if he has a mere 10 minutes of access to the phone. In our experiments, this meant that a 10 minute trial with file size equal to 5MB would cause the phone to attempt to download 2000MB (or 2GB) of data. Along with filling the space on the victim phone, this also monopolized the victim phone's network resources and impacted the other traffic on the network. Figure 19 shows the phone's network usage after one single 10 minute test with file size equal to 5MB was run.

5.5.2 Varying test length

Figure 20 shows results for the battery drain for various test lengths. We can see that the battery drain increases approximately linearly and is proportional to the amount of time the test ran for.

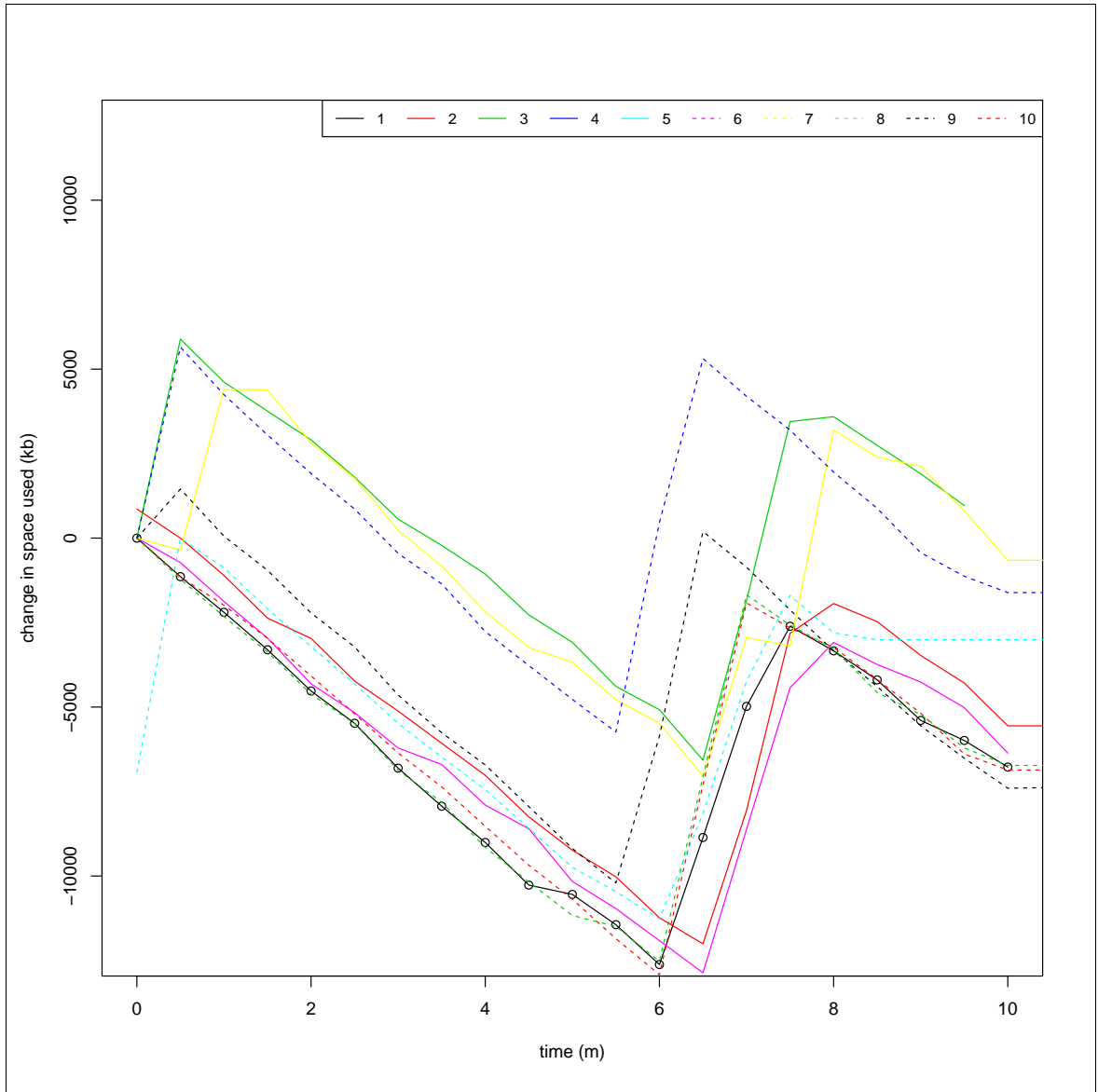


Figure 12: Storage consumption over 10 trials for file size=50kb

One interesting impact of running longer test lengths is the effect of a longer test on the phone's battery. Figure 21 shows the temperature of the victim phone's battery prior to and immediately following a 30 minute attack. With longer attacks, the battery will reach extremely high temperatures as shown in this example. This may lead to a negative impact on the phone's hardware due to the attack.

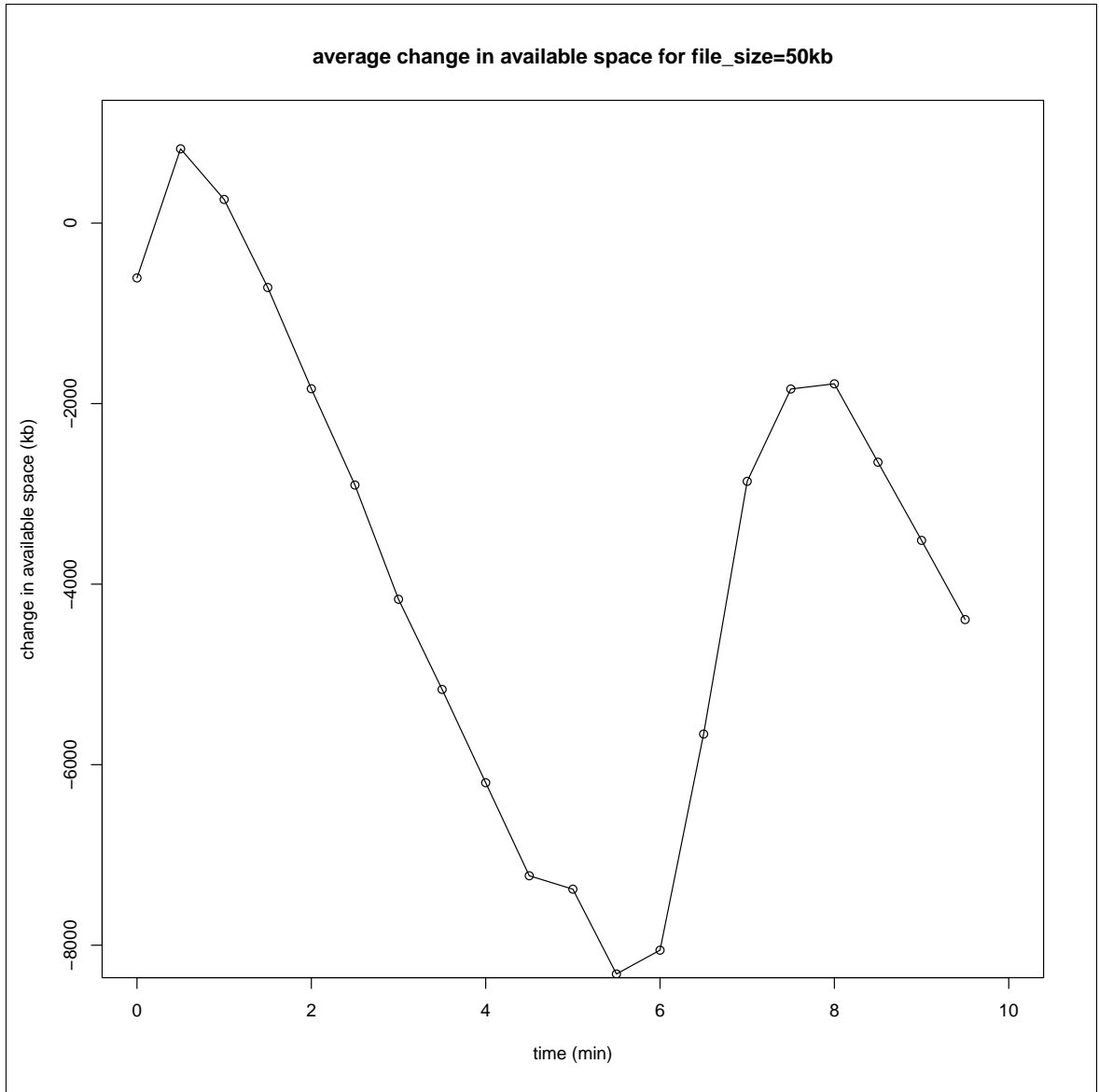


Figure 13: Average storage consumption for file size = 50kb

5.6 Discussion

We found that the automation system worked as a platform for launching a denial of service attack via NFC, with some limitations. The length of time necessary to perform a battery exhaustion attack via NFC may outweigh the value of the attack. On the other hand, it is unlikely that most users are carrying around phones with

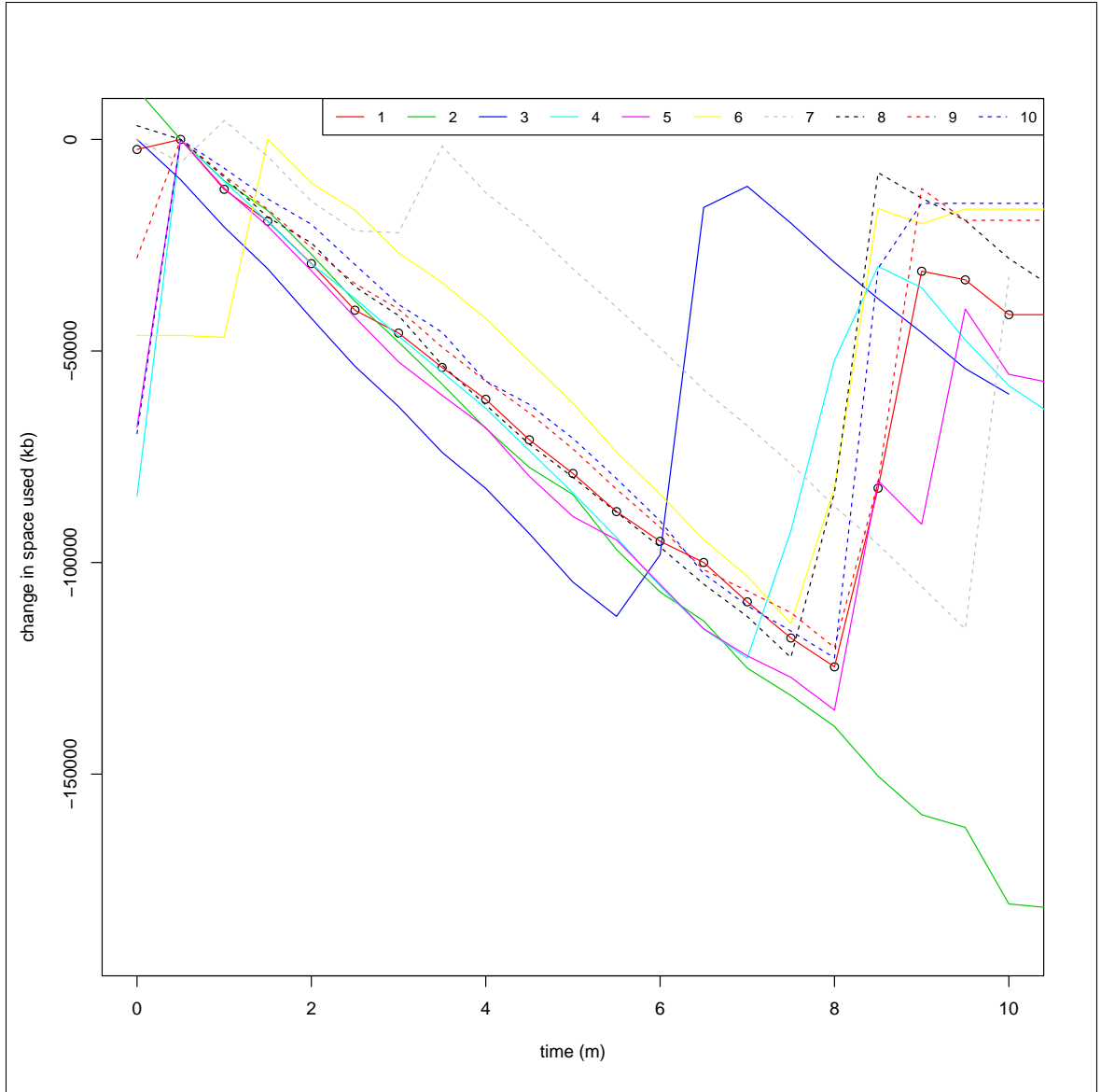


Figure 14: Storage consumption over 10 trials for file size=500kb

fully charged batteries. At a rate of 0.48 percent battery drain per minute, the attack may still pose a threat to partially charged phones.

The storage consumption attack may be the greater threat. We have shown that NFC can be used to kick off a massive number of file transfers, and that the transfers are not limited by size. Once the downloads have been started, they will continue until

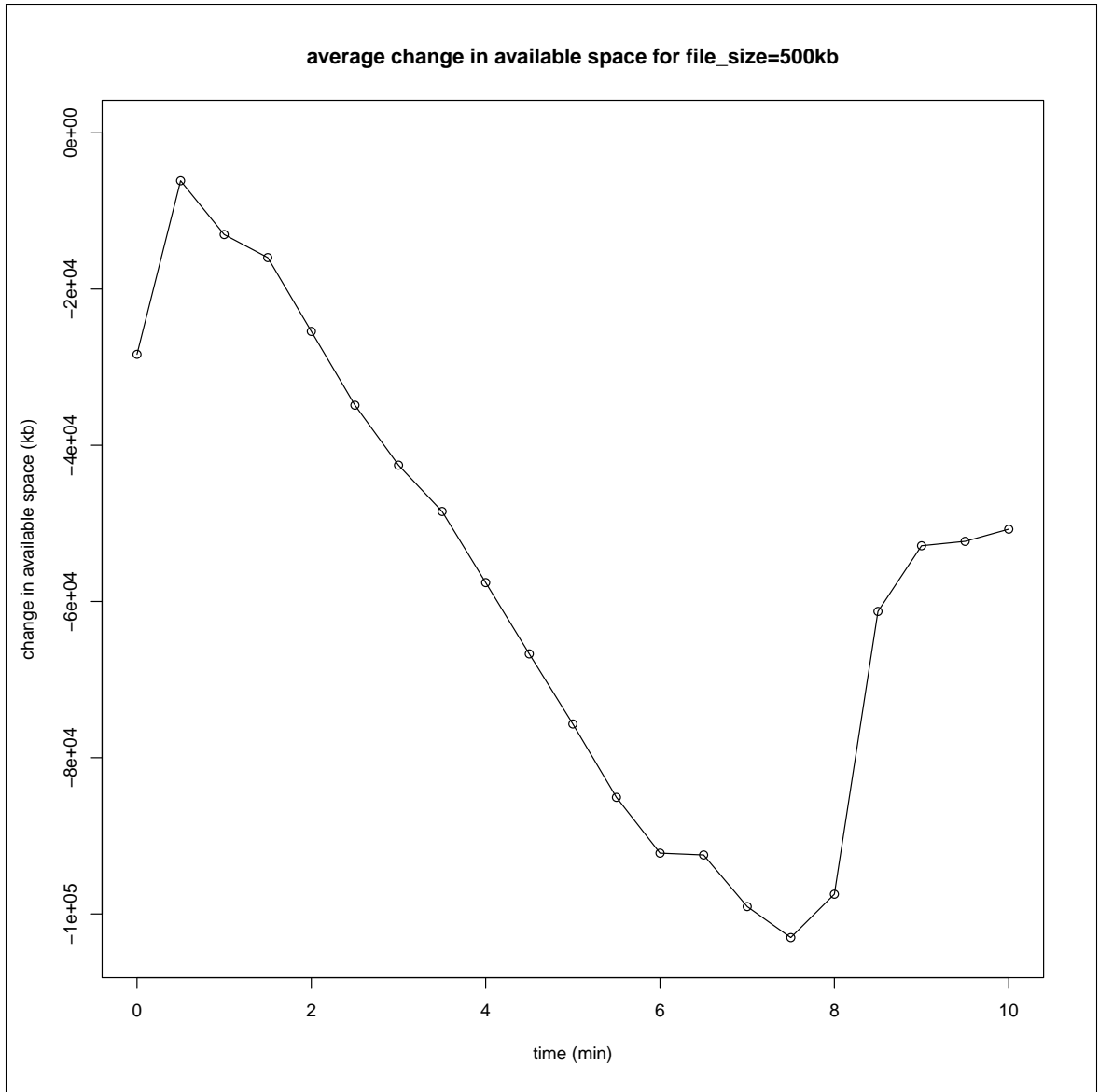


Figure 15: Average storage consumption for file size = 500kb

they are finished and cannot be cancelled by the user barring an extreme action such as a factory reset. The attack exploits the default behavior of the Android operating system with regards to handling NDEF records with the URI type containing links to certain file types such as .apk. This attack can be used to consume the entirety of the Galaxy Nexus' 13G of internal storage over time. Again, it is uncommon that the average cell phone customer would have a completely empty drive at the time of

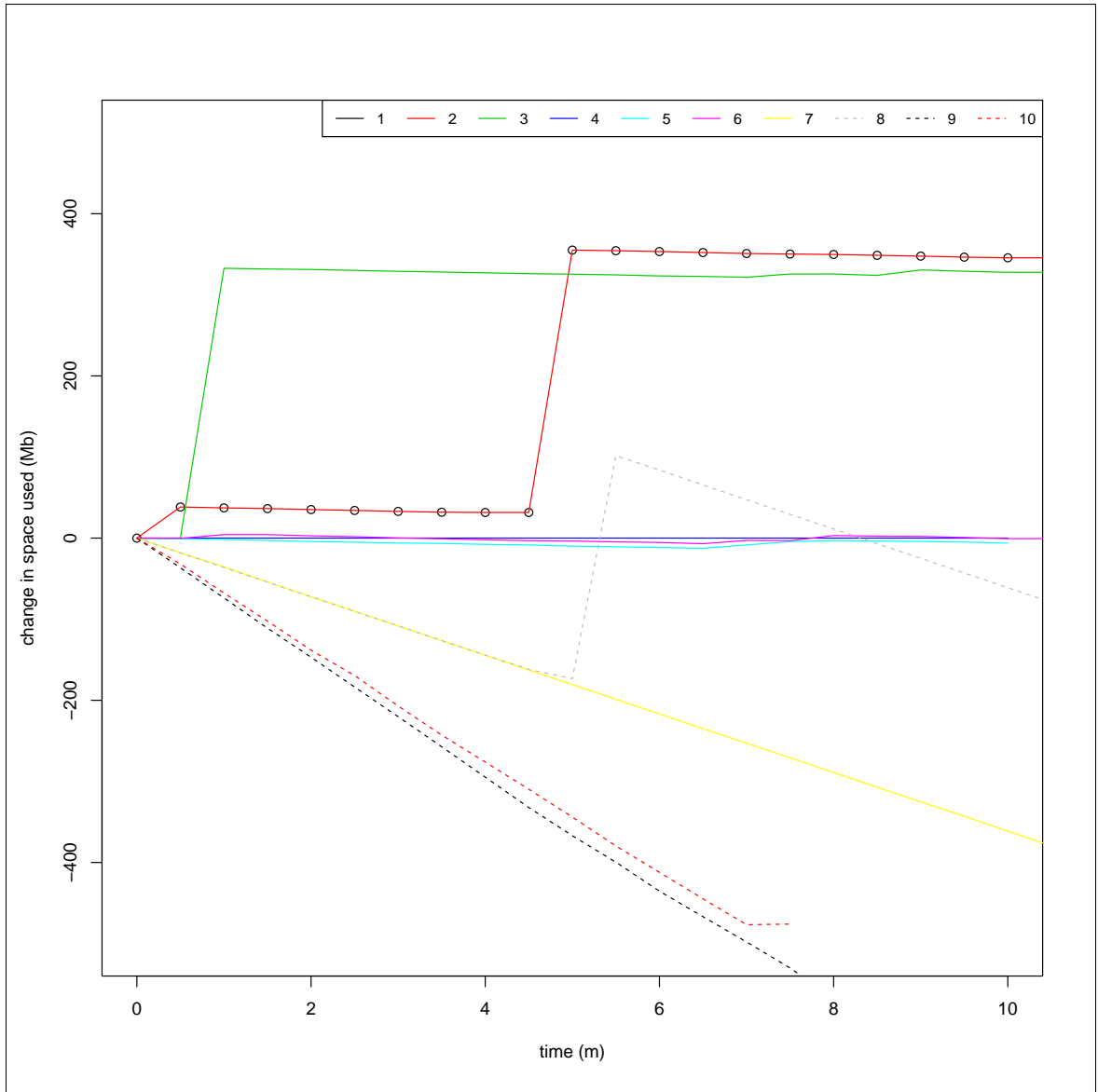


Figure 16: Storage consumption over 10 trials for file size=5Mb

an encounter with an attacker. This attack could devastate a phone with very little starting free space on it in a matter of minutes.

There are some potentially interesting ways to expand the attacks presented herein. Part of the impact of the storage consumption attack is the inconvenience it presents for the phone's user. In the best case scenario, the user notices the attack

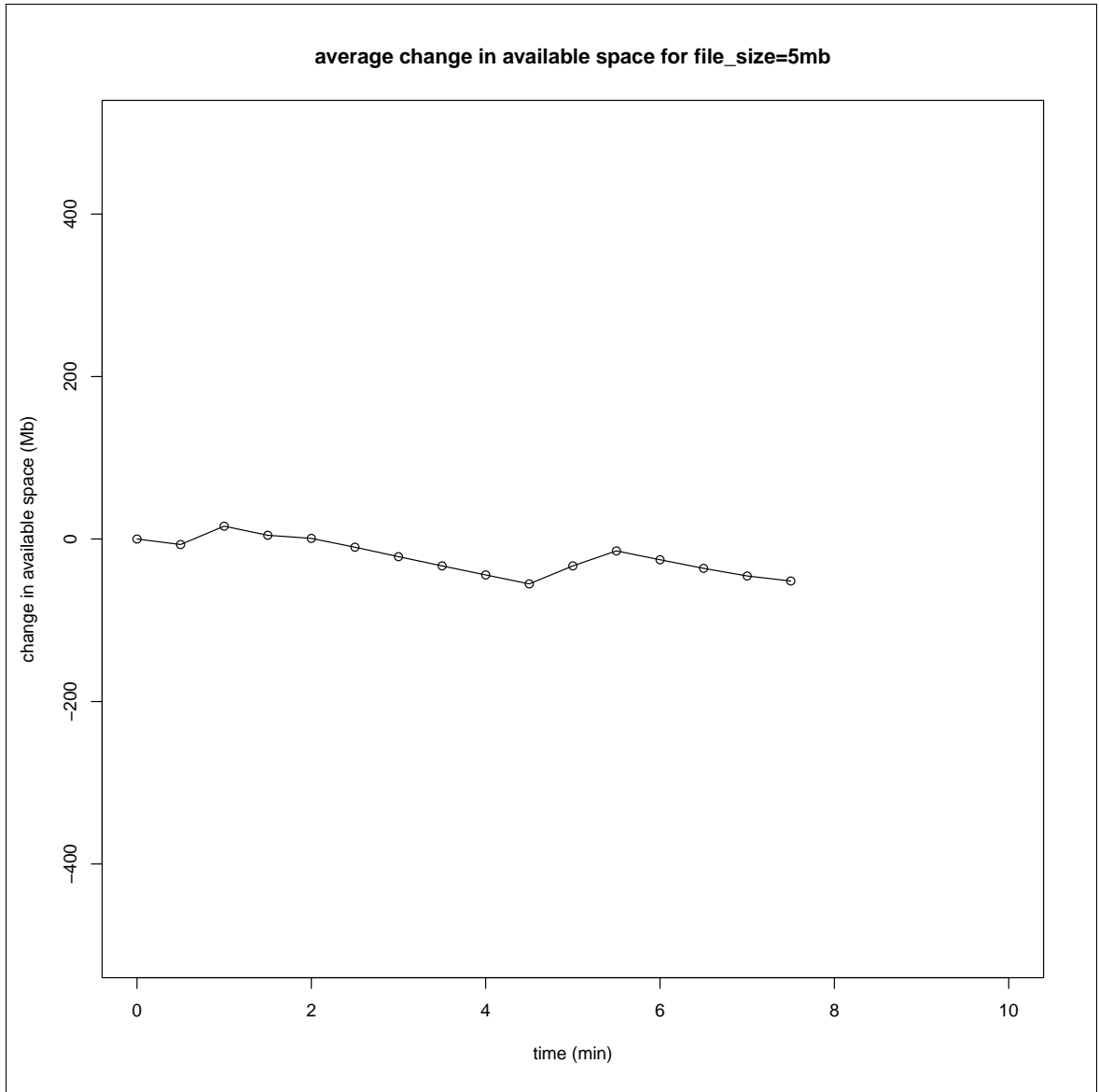


Figure 17: Average storage consumption for file size = 5Mb

and is able to move the phone out of range when only a few hundred file downloads have started. After the fact, the user must delete the downloaded files. In its present implementation, this is easy; since a fixed URI is used during the automation, all of the downloaded files have the same basic file name with a number appended. (This is how Android handles downloading a file with a duplicate name.) Consider the same attack, but the file names are randomized. In this case, they will be interleaved

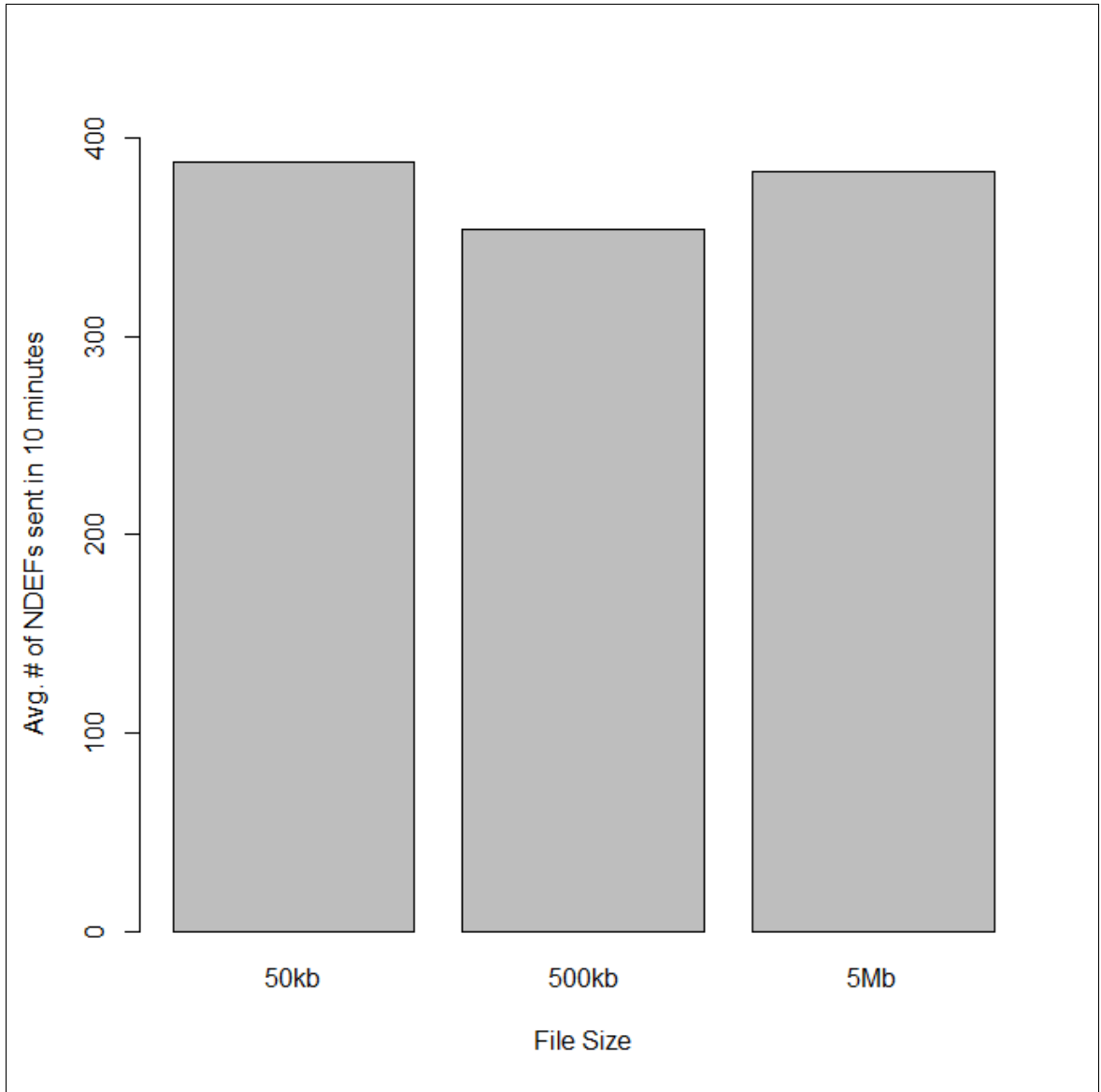


Figure 18: Number of NDEFs sent for various file sizes

with the user's legitimate files in the drive. Deleting the files from the attack without accidentally deleting legitimate data would become quite delicate work. In this way, it could lead to more data loss on the end user's part.

Another possible expansion of the attack would be to replace the dummy files used with a malicious URI. Figure 22 shows Android's notification screen; the user

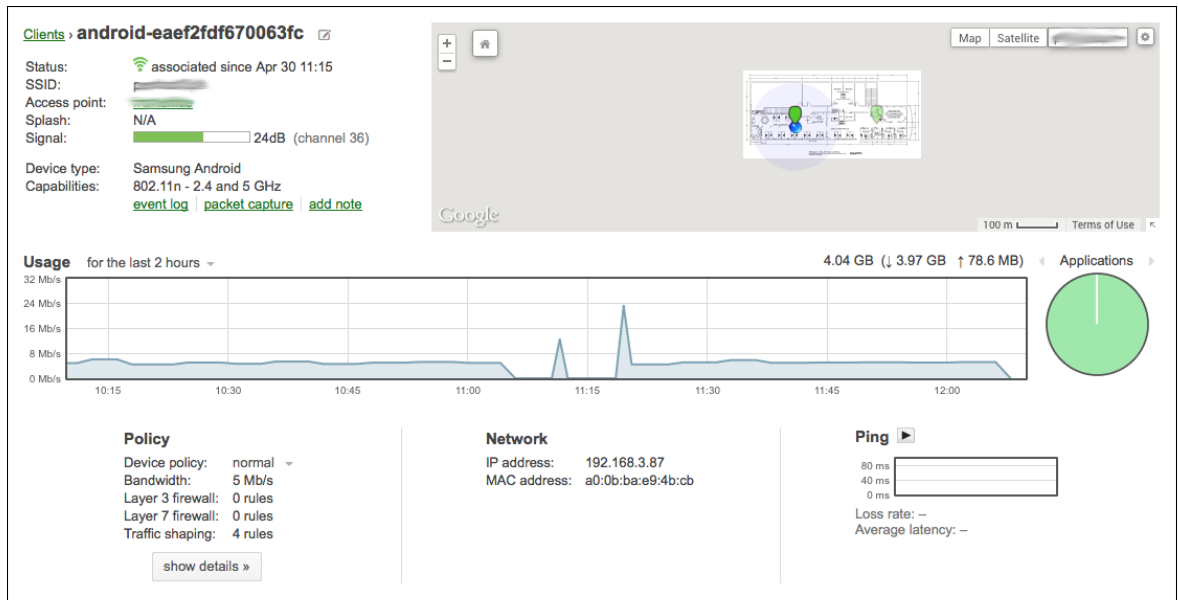


Figure 19: Victim phone's network usage for file size = 5MB

can clear the list of downloaded files by clicking the icon outlined in red. It is quite easy, while mass clearing notifications in Android, to accidentally click one of the .apks on the file list. Suppose these .apks contained an Android worm. The user accidentally clicks one while attempting to clear the hundreds of files downloaded during the attack. With a bit of social engineering flair, the user could be tricked into installing an .apk that, perhaps, promises to prevent this sort of attack in the future, and in doing so, open the door to an entirely different attack.

Finally, one interesting ramification of denial of service via NFC is the masking effect it has. To the server hosting the files used for the attack, the requests for URIs seem to come from the victim phone, despite the fact that they are generated by the attacking phone. The server has no way of knowing about the attacker's participation. This seems like a perfect starting place for malicious behavior.

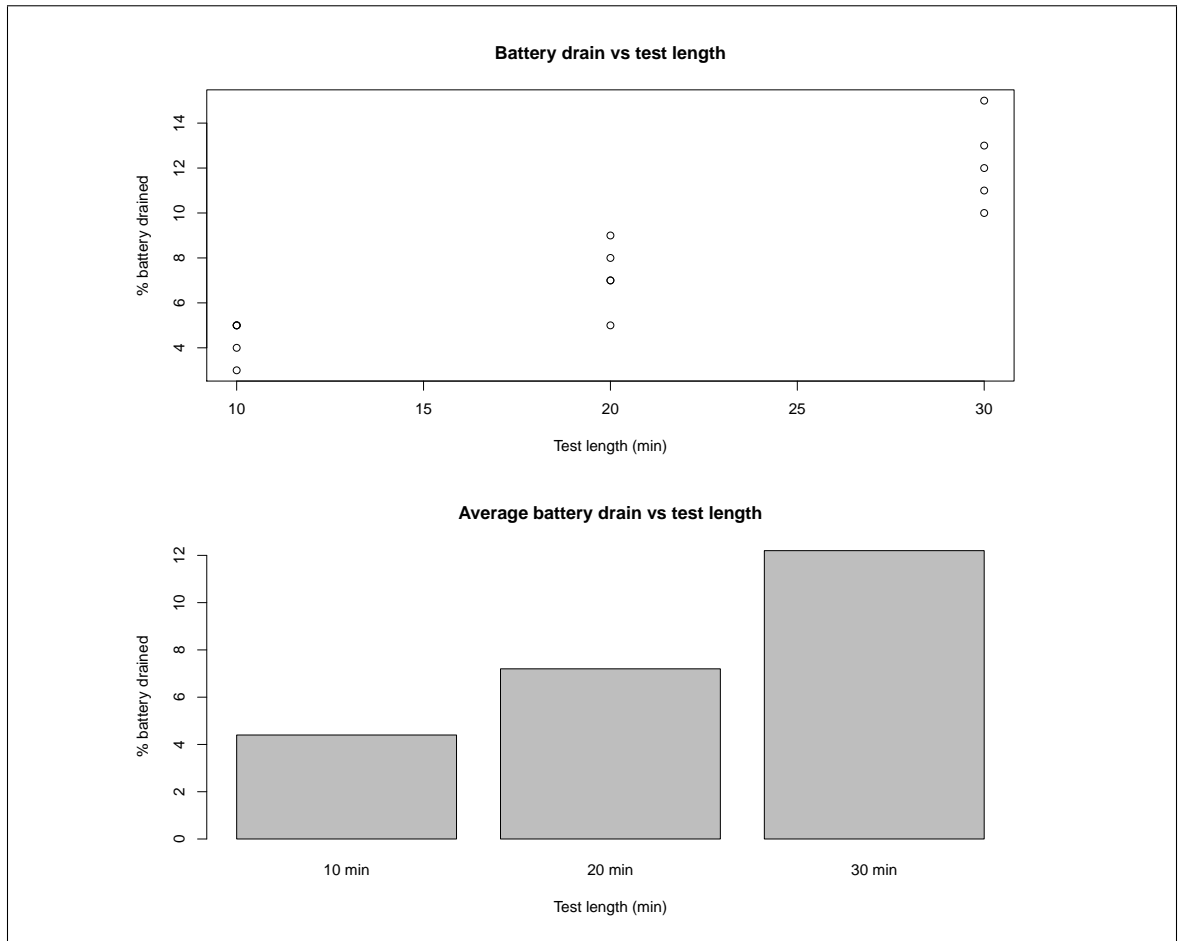
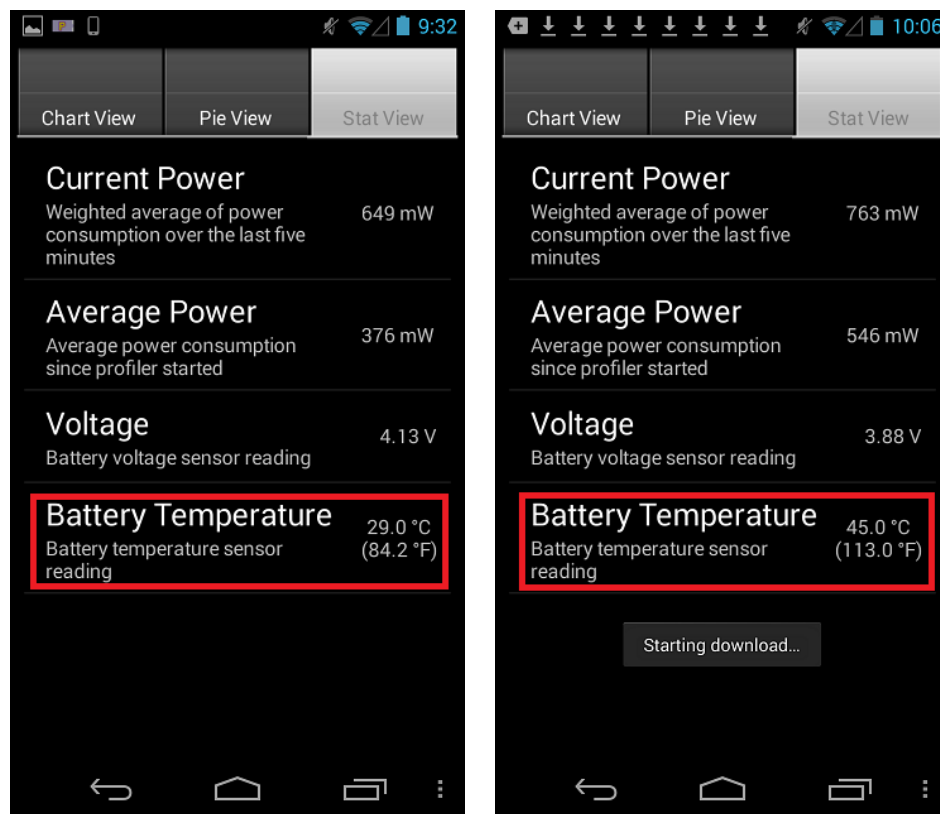


Figure 20: Battery drain versus attack length



(a) Before attack

(b) After attack

Figure 21: Battery temperature before and after 30 minutes of denial of service attack

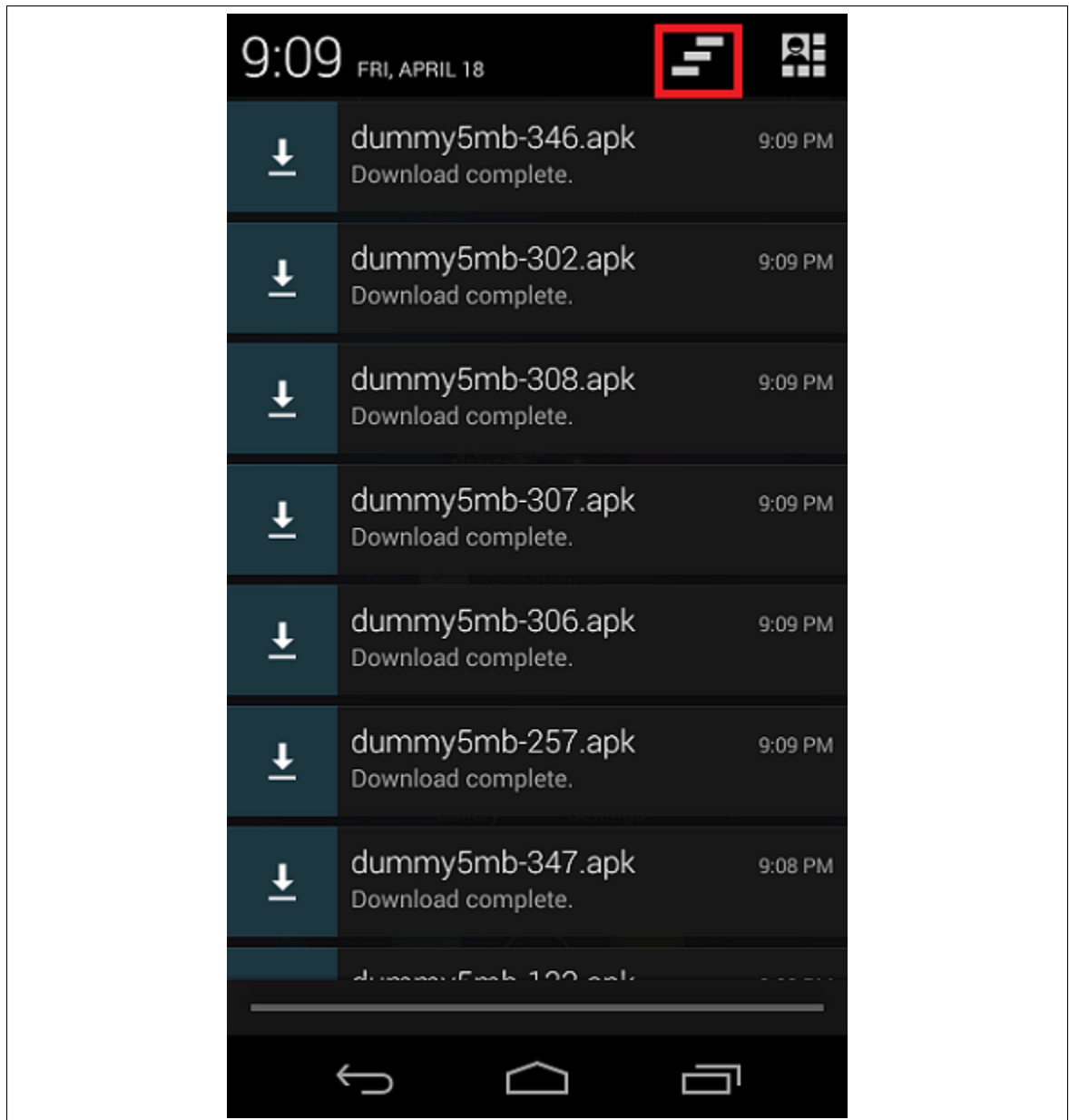


Figure 22: Android notifications screen

CHAPTER 6

Using Automation for Fuzz Testing

In this Chapter, we re-use the previously described automation framework for fuzz testing NFC. As in the denial of service attack, the basic idea is to repeatedly send requests from the source phone to the destination phone; however, unlike in the denial of service attack, the test inputs are carefully selected and the automation continues only until the input to be tested is exhausted.

6.1 Entities

As in Chapter 5, the phones used in the development of the fuzz testing application are two Samsung Galaxy Nexus GSM/HSPA+ phones. In this Chapter we will refer to them as the tester (source) phone and the testee (destination) phone.

The testee's purpose is to receive the fuzz testing input and handle it as a standard Android phone would. The testee is running stock Android Jellybean 4.3. It is important for the testee to run stock Android so that we can understand how the Android operating system handles NFC input by default; however, it would also be interesting to experiment with testees running custom Android ROMs or with user-installed NFC applications running. Certainly this could be done in the future using our automation framework. It will run as long as the NFC subsystem on the testee phone is standard.

One significant difference about the testee in fuzz testing versus the victim in denial of service is that we assume we have physical access to the victim phone before, during, and after testing, and we have permission to install our own appli-

cations on the victim phone. This is relevant for discovering interesting events that occur on the testee phone as a result of our fuzz testing. To monitor the results of the testing on the testee phone, we provide a simple application that monitors the log output on the testee phone and copies any interesting events to a file stored on the testee phone's SD card.

The tester's purpose is to generate the test input and run the test by sending the inputs to the testee. The tester phone is running our automation framework as described in Chapter 4. The tester is expected to be driven by human input at this time. It should be configured by hand.

An optional third entity is a monitor computer running ADB. If a third computer is used, both phones can be connected to it via USB and monitored using ADB. This may be useful for more in-depth debugging of problematic inputs, but will not be considered in the rest of this Chapter.

6.2 Interactions

Before the test is started, the monitoring application should be started on the testee phone. This must be done first so that no results are missed during the test run.

Once the monitoring application has been started, the tester launches the test using the previously described automation application. In most cases, the desired content source will be via remote file. This remote file is assumed to have previously generated test inputs in it. We also provide the option to generate a random file for input. This will generate 1000 test inputs using generation based fuzzing, e.g. totally random valid NDEFs. The most commonly selected test length for fuzz testing will be maximum time. This option will run the test until the end of the input is reached or

some maximum time is reached. The timeout assures that if there is some unexplained hang, such as a catastrophic NFC subsystem crash, the test will not continue to idle forever.

Once the test is started, it will run either until input is exhausted or the timeout is reached. While the test runs, the tester gathers statistics about the run and the testee monitors for any interesting results. It is configured to capture any log output containing an exception or the string NFC, or with a tag containing the string NFC. The former assures that exceptions will not be missed even if they are generated by applications NFC passes input to. The latter two encompass all log messages generated by the NFC subsystem.

After testing, the logs on both the tester phone and the testee phone can be examined to draw conclusions about the input. It may be useful to examine the testee log manually. We have also written a script, *testsummary.py*, to summarize the results of the run from the testee's perspective.

6.3 Assumptions

To simplify matters, we assume that the tester and testee phones are physically touching and will not be moved during the test. As in Chapter 5, the test will continue to run even if the phones are moved and will resume if they are separated and put back together again; however, doing this may generate unexpected results that do not relate to the fuzzing input but rather the physical change in device position.

Beyond that, our assumptions are greatly relaxed for the fuzz testing. The testee phone does not necessarily need to be running stock Jellybean. Both phones still need to be NFC-capable and have NFC enabled.

6.4 Generating Random Inputs

Previous research has suggested that some tools exist to generate random NDEFs as fuzz testing inputs; however, none of these tools appears to be publicly available at this time. As a result, we have provided a simple function within the automation framework for generating random NDEFs. It is based around the NDEF message format and results in the construction of an *NdefMessage* [3].

The random inputs generated by the program are generated using a series of functions in the automation application. The basic idea is this: we form a dependency tree of the various fields in an NDEF record, then follow the tree from top to bottom. The NDEF fields are described in Table 1 in Chapter 1. First a random number is generated to ascertain the TNF of the NDEF to be generated. Then, depending on the TNF chosen, a Type field may be generated by generating a random number and picking the corresponding Type out of a list. If a Type is selected, the corresponding Type Length is used for that field. Table 8 summarizes the TNF options, and 9 summarizes the types generated by our input generation script. (Note that the types of NDEF records generated by our input generation script are a subset of all of the possible NDEF record combinations.)

MB and ME (Message Begin and Message End) are always set to 1 in our generation as we only generate inputs containing one record per message. Similarly, the CF (Chunk Flag) flag is always set to 0 as Android does not use chunking for NFC.

6.5 Fuzz Test Implementation

Our implementation of fuzz testing using the same automation framework as our denial of service attack, but to a different end. Rather than having an attacker phone flood a victim phone with a repeated message, the messages to be sent are purposefully

Value	Type	Usage
0x00	Empty	Payload has length zero, contents are empty
0x01	NFC Forum well-known type	Contains one of the most commonly used record types
0x02	Media type	Contains a file with a MIME type specified
0x03	Absolute URI	Contains a URI
0x04	NFC Forum external	Contains a type of record defined by an organization other than the NFC Forum
0x05	Unknown	Type not described in the first 5 types; may be a malformed record
0x06	Unchanged	Used to mark a record that contains a continuation of a previous record
0x07	Reserved	Reserved for future use

Table 8: Type-Name-Format (TNF) values

TNF	Type	Meaning
0x00	Empty	NDEF message containing one empty record
0x01	Text	Plain text record
0x01	URI	Contains a URI
0x03	Absolute URI	Contains a URI

Table 9: Available NDEF types

chosen to test the behavior of NFC. Both phones are assumed to be complicit in the test. As before, the tester (source) phone launches the test to send input to the testee (destination) phone. The content source can either be an independently generated set of inputs, specified in the remote URI field, or a randomly generated (by our framework) set of inputs. The selected test length should be max time, as this will only run to the end of input or until a timeout is hit.

When the source is specified as a remote URI, the file should be in the format described in Appendix A.4. If the content source is set to random, the application

will call a script that generates 1000 random NDEF messages, then stores them in a file on the SD card. The input generation is described in Section 6.4. This file is then used as the input source for the test.

We conducted two experiments to test the capabilities of the framework with respect to fuzz testing:

1. **Input generation** First, we tested the ability of the application to successfully generate valid, random NDEF messages. To do this, we used the random file content source option to generate 1000 random NDEFs. Then we monitored the logcat output on both the tester and testee phones for any unusual behavior that would indicate an incorrectly structured NDEF.
2. **Testing a large set of inputs** After we had confirmed the validity of our NDEF generation code, we proceeded to generate a large number of inputs using it. We generated 1,000 random NDEFs and monitored for errors. Our experiments in Chapter 5 indicated that about 40 NDEF messages can be sent per minute; thus it should be possible to run the entire set of fuzzing inputs in around 2 hours.

6.6 Initial Results

We were able to generate countless random NDEFs with our framework and did not encounter any exceptions on the tester side or the testee side except for UI complaints. On the tester phone, the UI thread would occasionally throw an exception indicating that the program was taking too long elsewhere; this is likely related to the amount of time needed to generate the random messages.

After we confirmed that we could create a small set of messages without issue,

we let the test run until the battery ran down (approximately 3 hours) on the testee phone. During that time, the tester phone repeatedly sent randomly generated NDEFs to the testee phone. Both phones were able to tolerate the load without throwing any unusual exceptions.

6.7 Discussion

The automation framework has been shown to be a viable option for fuzz testing NFC. It provides an easy mechanism for testing large sets of NDEF messages. NDEF messages can either be provided in a remote file or randomly generated during program execution. The current fuzzing generation feature of the framework is rudimentary at best and does not allow for nuanced testing; however, we have proven that the platform can be used for fuzz testing. Possible future work could investigate better structuring of random NDEFs.

CHAPTER 7

Future Work

At the end of the day, there are still a number of topics related to this project that would be interesting to explore further, including improvement of NDEF fuzz testing input generation, testing against a wider range of NFC phones and using NFC as a sort of tunnel to hide an attacker's identity.

Across existing NFC fuzz testing research, a recurring theme is the lack of a good solution for generating input to feed into NFC. We have provided a rudimentary system for generating random NDEF records. This system could be expanded upon. One significant potential improvement would be to add mutation based input generation along with the generation based input creation we have provided.

Implementing our automation framework to work on real hardware gave us many interesting insights into the practicalities associated with developing for modern NFC. Using actual hardware exposes timing and synchronization issues masked by the emulator, and reveals physical characteristics that may complicate or prevent NFC transmission. We had limited access to NFC-capable Android phones for this project, but expect that our automation framework would work on any Android phones with NFC capabilities running JellyBean. Future work could test using other models of phones, including other vendors' implementations, for the source and destination phones.

In this project, we explored using automation to perform denial of service and fuzzing. Another possible sinister use case for it would be to send commands from a malicious phone to a victim phone, who would then execute the command. For instance, the malicious phone could generate inputs to flood a particularly server

with requests, then automate sending those inputs to a victim phone. The victim phone would then relay those messages to the server. From the server’s perspective, the victim would seem to be the source of the attack. In this way, the transient nature of physical NFC contact might even be a boon. The attacker could move from place to place, shunting his requests through whatever innocent phones he came across.

Another interesting application for the existing framework would be to investigate denial of service via passive tags. The idea is this: the victim phone’s user is somehow convinced to install the automation framework. Perhaps it is bundled along with legitimate looking software as many Android viruses are. Once installed on the victim’s phone, the automation application goes into an infinite loop; it runs essentially with test until failure selected. But rather than seeking to constantly send, as the automation framework does in our denial of service attack, the victim phone will constantly advertise itself as a receiver. Should it come in contact with another device or even a passive tag, it will get stuck in a cycle of attempting to read from that entity, and in doing so may basically perform denial of service upon itself.

One assumption we made in our denial of service work is that the attacker would have a means of assuring that the victim’s phone screen would stay on. Recall that the screen must be on and unlocked for NFC transactions to occur. One significant improvement to the implemented attack would be to add the ability to trigger the victim’s screen to stay awake. In [25], Miller shows a method for unlocking the screen of a victim phone using a dialer and the victim phone’s number. It might be possible to implement such a dialer within the application.

The files transferred during our tests were benign in nature. We briefly discussed the impact of using the framework to deliver malicious files in Section 5.6. In general, malicious application installation in Android relies on some social engineering.

Drawing from that, it is possible to trick users into installing a malicious application when they are in fact trying to delete it, by relying upon misclicks when clearing the file notification screen.

Similarly, the NDEFs transmitted could contain URIs to sites containing malicious, illegal, or unsettling content. The impact of this sort of attack is more difficult to measure than a battery exhaustion attack, of course. But this attack could have a real world impact in certain scenarios due to the misleading audit trail for NFC. Consider this scenario: an attacker gains brief physical access to a smartphone belonging to a public official. He uses an NFC-based attack to direct the official's phone to sites containing inappropriate content. Now there's evidence on the server side that the official accessed this material, but no trail back to where the requests really originated from (the attacker's phone).

It would also be interesting to look at combining NFC with scripting-based browser attacks. Again, this could be extremely useful for malicious muck raking. It seems possible to use NDEFs to pass URIs that post to web forms. The attacker would need to identify a webform suitable for the attack, one that passes the data for the form in a query string that can be spoofed. Then, if he knows the identity of the victim, he can pass in parameters to the web form to post inflammatory comments that, by all accounts, appear to be "from" the victim.

Our experiments investigated the functional operation range of NFC and showed that until the transmission barrier is reached, approximately the same number of messages can be sent in a given time period at any distance. It would be interesting to examine whether or not the distance between the devices during operation impacts the battery drain on the victim phone during the attack.

Another variable that could be tested in our experimental setup is the time between NFC resets. We checked for the need to reset every 5 seconds. This value seemed to work well. The number of resets tended to be very low compared to the number of messages successfully sent over any given interval, implying that there were rarely unnecessary resets. But perhaps 5 seconds is unnecessarily frequent, and the same hangs could be prevented by checking every 10 or 20 seconds. There is some overhead associated with checking for the need to reset, so the number of messages successfully sent over an interval might increase if there was less frequent reset checking.

Finally, we should consider a means of preventing the denial of service as performed in this paper. One of the key components of this attack is the behavior of the default Android browser. It could be modified to drastically reduce the efficacy of this attack. For instance, it could be altered to only allow a certain number of files with the same name to be downloaded. Or, it could be altered to only allow a certain number of outstanding download requests. Certainly it could be changed to allow users to cancel their outstanding downloads simultaneously; even outside of preventing this attack, that would improve the usability of the device.

CHAPTER 8

Conclusions

In this work, we presented a motivation for and a detailed description of a novel method of automating messages between two NFC-enabled cell phones (Chapters 3 and 4).

We then applied this framework to two separate problems. First, we used the framework to perform a denial of service attack against a cell phone, targeting both the battery on the phone and the internal storage on the phone (Chapter 5). We were able to successfully drain the battery of the phone and consume its entire storage space. We also measured the impact of the attack over shorter time intervals to draw some conclusions about the potential for shorter, more practical denial of service attacks.

Second, we used the framework as a means of performing basic fuzz testing for NFC (Chapter 6). We showed how the exact same framework could be applied with a different set of settings to reach this end. We provided some means of generating input for fuzz testing and showed that our scripts resulted in valid NDEF messages. We then showed how our system could be used for sending these messages and some guidelines on monitoring results on the destination phone's end.

And finally, we concluded with some ideas on expanding our research in future projects (Chapter 7).

This work shows that NFC automation is a viable, implementable technology. It shows some potential positive use cases for NFC automation as well as some potential pitfalls.

LIST OF REFERENCES

- [1] Amini, P., and Protnoy, A. Sulley Fuzzing Framework. <http://www.fuzzing.org/wp-content/SulleyManual.pdf>
- [2] Android Developers. NdefRecord. <http://developer.android.com/reference/android/nfc/NdefRecord.html>
- [3] Android Developers. NdefMessage. developer.android.com/reference/android/nfc/NdefMessage.html
- [4] Android Developers. Near Field Communication. <http://developer.android.com/guide/topics/connectivity/nfc/index.html>
- [5] Android Developers. Reading and Writing Logs. <http://developer.android.com/tools/debugging/debugging-log.html>
- [6] Boden, R. One in Three Smartphones Now Comes with NFC. *NFC World*. 1 August 2013. <http://www.nfcworld.com/2013/08/01/325283/one-in-three-smartphones-now-comes-with-nfc/>
- [7] Bose, A. and Shin, K.G. On Mobile Viruses Exploiting Messaging and Bluetooth Services. *Securecomm and Workshops 2006*. 28 August–1 September 2006. doi:10.1109/SECCOMW.2006.359562
- [8] Bradshaw, S. An Introduction to Fuzzing: Using fuzzers (SPIKE) to find vulnerabilities. <http://resources.infosecinstitute.com/intro-to-fuzzing/>
- [9] Caney, R.; Dorros, C.; Kennedy, S.; Owens, G.; and Tague, P. Mobile Pick-pocketing: Exfiltration of Sensitive Data through NFC-Enabled Mobile Phones. Technical Report. *Carnegie Mellon University*. 5 December 2013.
- [10] Carroll, A. An Analysis of Power Consumption in a Smartphone. *Proceedings of the 2010 USENIX conference*. 2010.
- [11] Clark, Sarah. Wired publishes Lexus NFC ad. *NFC World*. <http://www.nfcworld.com/2012/03/22/314720/wired-publishes-lexus-nfc-ad/>
- [12] Cooper, D. Barclays releases PayTag: the NFC card you glue to your phone. *Engadget*. www.engadget.com/2012/04/19/barclays-paytag/
- [13] Coskun, V., Ozdenizci, B., and Ok, K. A Survey on Near Field Communication (NFC) Technology. *Wireless Personal Communications*. 71(3):2259–2294. August 2013. doi:10.1007/s11277-012-0935-5

- [14] dd. Linux man page. <http://linux.die.net/man/1/dd>
- [15] df. Linux man page. <http://linux.die.net/man/1/df>
- [16] DeMott, Jared. The evolving art of fuzzing. *DEFCON 14*. 5 August 2006.
- [17] Dunning, J.P. Taming the Blue Beast: A Survey of Bluetooth Based Threats. *IEEE Security & Privacy*. 8(2):20–27. March-April 2010. doi:10.1109/MSP.2010.3
- [18] Gligor, V.D. A Note on Denial-of-Service in Operating Systems. *IEEE Transactions on Software Engineering*. 10(3):320–324. May 1984. doi:10.1109/TSE.1984.5010241
- [19] Haselsteiner, E. and Breitfuß, K. Security in near field communication (NFC). *Workshop on RFID Security (RFIDSec)*. 2006.
- [20] Information technology - Telecommunications and information exchange between systems - Near Field Communication - Interface and Protocol (NFCIP-1). *Draft international standard ISO/IEC 18092:2013*.
- [21] Keep Screen On Free. ByOne Coder. *GooglePlay store*. <https://play.google.com/store/apps/details?id=com.byonetech.android.screenon>
- [22] Madlmayr, G.; Langer, J.; Kantner, C. and Scharinger, J. NFC Devices: Security and Privacy. *International Conference on Availability, Reliability and Security, 2008 (ARES '08)*. 4-7 March 2008. doi:10.1109/ARES.2008.105
- [23] Maklia, T., Taimisto, J. and Vuontisjarvi, M. Fuzzing Bluetooth: Crash-testing bluetooth-enabled devices. http://www.codenomicon.com/resources/whitepapers/codenomicon_wp_Fuzzing_Bluetooth_20110919.pdf
- [24] Martin, T.; Hsiao, M.; Ha, D. S.; and Krishnaswami, J. Denial-of-service attacks on battery-powered mobile computers. *IEEE Annual Conference on Pervasive Computing and Communications 2004 (PerCom '04)*. 14-17 March 2004. doi:10.1109/PERCOM.2004.1276868
- [25] Miller, C. Exploring the NFC Attack Surface. *In proceedings of Black-Hat 2012*. http://media.blackhat.com/bh-us-12/Briefings/C_Miller/BH_US_12_Miller_NFC_attack_surface_WP.pdf
- [26] Moyers, B.R.; Dunning, J.P.; Marchany, R.C.; and Tront, J.G. Effects of Wi-Fi and Bluetooth Battery Exhaustion Attacks on Mobile Devices. *Hawaii International Conference on System Sciences 2010 (HICSS '10)*. 5-8 January 2010. doi:10.1109/HICSS.2010.170

- [27] Mulliner, C. Vulnerability Analysis and Attacks on NFC-Enabled Mobile Phones. *International Conference on Availability, Reliability and Security, 2009 (ARES '09)*. 16-19 March 2009. doi:10.1109/ARES.2009.46
- [28] Mulliner, C. Attacking NFC mobile phones. *EUSecWest 2008*. May 2008.
- [29] Mulliner, C. Hacking NFC and NDEF. *NinjaCon 11 (B-Sides)*. 18 June 2011.
- [30] NFC Forum. NFC Record Type Definition (RTD) Technical Specification. http://www.nfc-forum.org/specs/spec_list/
- [31] NFC Forum. NFC Data Exchange Format (NDEF) Technical Specification. http://www.nfc-forum.org/specs/spec_list/
- [32] NFC Forum. What is NFC? About the Technology. <http://nfc-forum.org/what-is-nfc/about-the-technology/>
- [33] Pelechrinis, K.; Iliofotou, M.; and Krishnamurthy, S.V. Denial of Service Attacks in Wireless Networks: The Case of Jammers. *IEEE Communications Surveys & Tutorials*. 13(2):245–257. Second Quarter 2011. doi:10.1109/SURV.2011.0411110.00022
- [34] Racic, R.; Ma, D.; and Chen, H. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. *Securecomm and Workshops 2006*. 28 August–1 September 2006. doi:10.1109/SECCOMW.2006.359550
- [35] Robotium: The World's leading Android test automation framework. <http://code.google.com/p/robotium/>
- [36] Roland, M.; Langer, J.; and Scharinger, J. Practical Attack Scenarios on Secure Element-Enabled Mobile Devices. *International Workshop on Near Field Communication (NFC) 2012*. 13 March 2012. doi:10.1109/NFC.2012.10
- [37] Roland, M.; Langer, J.; and Scharinger, J. Applying relay attacks to Google Wallet. *International Workshop on Near Field Communication (NFC) 2013*. 5 Feb. 2013 doi:10.1109/NFC.2013.6482441
- [38] Sosonoski, D. Java programming dynamics, Part 2: Introducing reflection. *IBM developerWorks*. 3 June 2003. <http://www.ibm.com/developerworks/library/j-dyn0603/>
- [39] Stirparo, P. A Fuzzing Framework for the Security Evaluation of NDEF Message Format. *5th International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN '13)*. pp.165–170. June 5-7 2013. doi:10.1109/CICSYN.2013.58

- [40] Sutton, M.; Greene, A.; and Amini, P. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [41] Touch to beam UI. Digital image. *IntoMobile*. <http://www.intomobile.com/2011/12/01/closer-look-android-beam-samsung-galaxy-nexus/>
- [42] Van Damme, G.; Wouters, K. and Preneel, B. Practical Experiences with NFC Security on Mobile Phones. *In proceedings of the Workshop on RFID Security and Privacy 2009 (RFIDSec '09)*..
- [43] Verdult, R.; Gans, G.; and Garcia, F. A toolbox for RFID protocol analysis. *IEEE Fourth International EURASIP Workshop on RFID Technology (EURASIP RFID), 2012*. 28-29 September 2012.
- [44] Verdult, R.; Kooman, F. Practical Attacks on NFC Enabled Cell Phones. *3rd International Workshop on Near Field Communication (NFC) 2011*. pp.77–82. 22 February 2011. doi:10.1109/NFC.2011.16
- [45] Vidas, T.; Zhang, C.; and Christin, N. Toward a general collection methodology for Android devices. *In proceedings of the 11th Annual Digital Forensics Research Conference (DFRWS 2011)*. 8:S14–S24. August 2011.
- [46] Von Behren, R. and Wall, J. Coming soon: make your phone your wallet. 26 May 2011. *Google: Official Blog*. <http://googleblog.blogspot.com/2011/05/coming-soon-make-your-phone-your-wallet.html>
- [47] Wiedermann, N. Fuzzing-to-go: A test framework for Android devices. *Master's thesis*. Technische Universität München, 2013.
- [48] WugFresh. Nexus Root Toolkit v1.8.2. <http://www.wugfresh.com/nrt/>
- [49] Yang, Z. PowerTutor-A Power Monitor for Android-Based Mobile Platforms. <http://ziyang.eecs.umich.edu/projects/powertutor/index.html>
- [50] Zhang, L. NFC Application Development on Android with Case Studies. *Intel Developer Zone*. 30 July 2013. <http://software.intel.com/en-us/articles/nfc-application-development-on-android>

APPENDIX A

Practicalities

A.1 Physical Phone Setup for Experiments

Unless specified otherwise, during all of the experiments, the source phone and destination phone were literally touching. One phone lay on top of the other one. An interesting quirk of Android’s NFC implementation is that NFC transactions cause the destination phone to vibrate. The force of the vibration is enough that, over time, the top phone is likely to slide off of the bottom phone. As discussed in Section 5.3.1, the phones need to stay quite close to each other and a horizontal move can cause the connection to be lost. An extremely simple yet effective solution to this problem is to rubber band the two phones together during testing.

It is also worth noting that by default, NFC transactions cause both of the phones to vibrate during the transaction. There is no direct way to disable this (though one can always rebuild the Android source to disable it). The vibration generated by the Galaxy Nexus is quite loud. Wrapping the rubber-banded phones in a towel dampens the sound considerably and does not appear to impede connectivity in any way; however it should not be done when testing the impact of an attack on the battery’s temperature as the inclusion of the towel may impact the amount of heat retained.

A.2 Rooting the Galaxy Nexus

In order to install our automation application as a system application, we needed to gain superuser privileges on the source phone. The only method of doing this in Android is to root the phone. There are several possible ways to go about rooting an

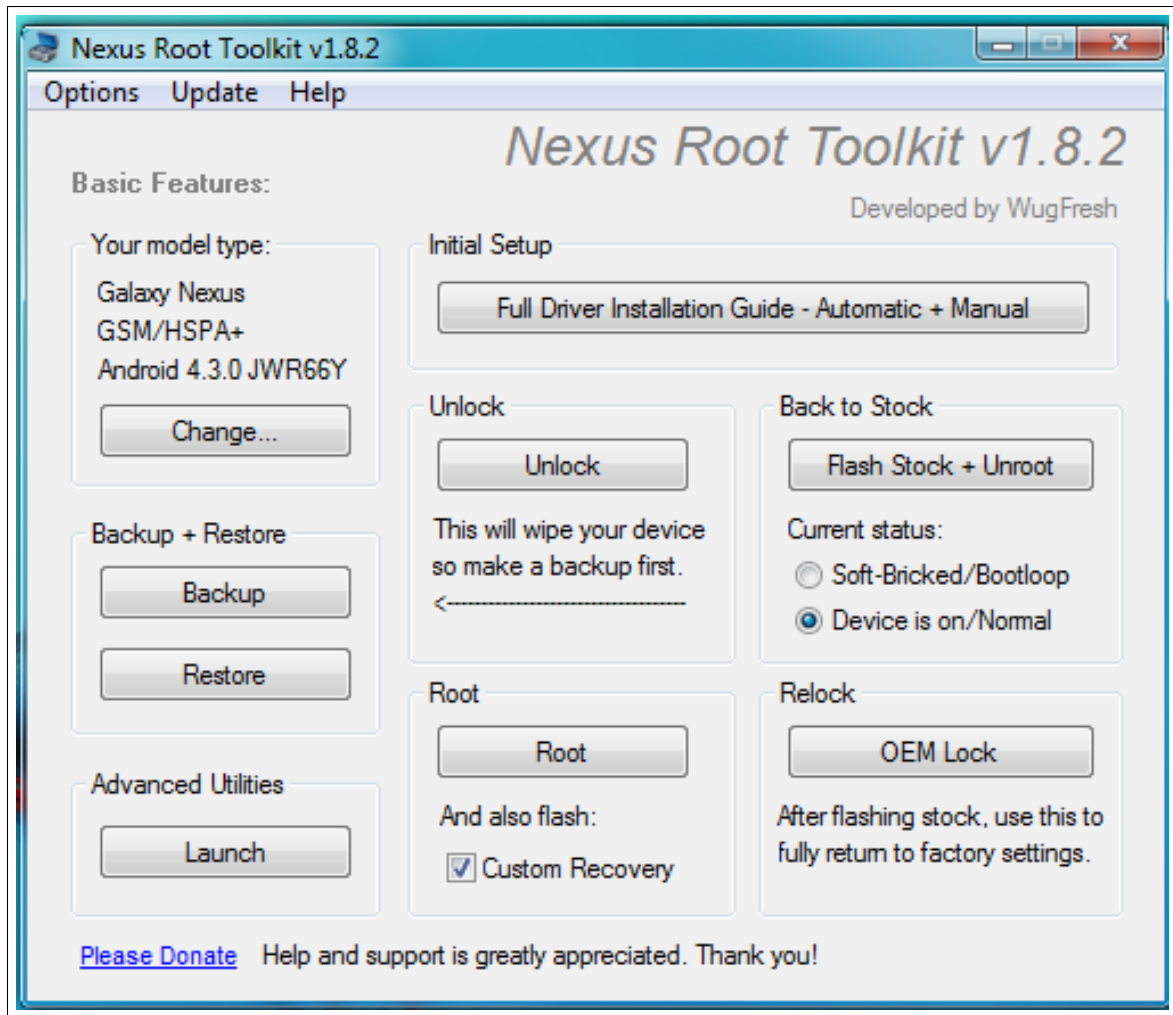


Figure A.23: Nexus Root Toolkit UI

Android phone. We chose to use the Nexus Root Toolkit v1.8.2 by WugFresh [48]. In this Section, we will give a brief description of the NRT and explain its usage.

The Nexus Root Toolkit (NRT) is a piece of software primarily designed to allow for simple unlocking and rooting of an Android phone, but also includes some useful additional features such as being able to backup the phone, wipe the phone's data, or restore the phone to factory defaults with great ease.

Figure A.23 shows the interface for the NRT.

We found all of the basic features useful at some point in the project. The Initial Setup simplifies identifying and installing the correct drivers for communicating with the phone via Windows, a non-trivial task. Unlock performs the same action that usually requires the user to reboot into the fastboot screen and manually make a change. Once the device has been unlocked, the Root button takes the user through a mostly automated rooting process. Back to Stock was a relief to have during development, as the phone was put into a boot loop or “bricked” state at least once. Backup and Restore was similarly useful throughout development.

A.3 Using `dd` to Generate Files with Random Contents

For the storage space consumption attacks in Section , we generated three files with random contents for the phone to download from the remote server. We did so using the Unix `dd` command [14]. This command, as used here, takes the desired size of a file as an input and generates a file of that size containing randomly selected bytes. The command can be entered like this:

```
if=of=filename bs=size_in_bytes count=1
```

A.4 Input File Format

When using remote (user-specified) files for the content source for automation, it is important that the files conform to the application’s expectations. Each line in the input file should represent a separate NDEF record to be sent in an individual NDEF message.

There are two valid types of lines: URIs and raw NDEF records. URIs are the simpler case; any valid URI may be used. Lines containing URIs should be marked with `uri:` prior to the actual URI. This is a correct example of a URI line in an input

file:

uri:http:

www.reddit.com °

This is an *incorrect* example of a URI line in an input file, lacking a protocol:

uri:www.reddit.com °

The second type of acceptable input line is a raw NDEF record. In this case, a file containing the exact bytes to describe the NDEF record is provided. The byte values should be specified in a way that builds a coherent NDEF record per the format's specifications [31]. The automation application does minimal input checking and may not be able to recover from malformed NDEFs gracefully.

A.5 Interesting Failures that Occurred During Testing

Several unusual behaviors appeared during our tests. For the purposes of our experiments, these results needed to be thrown out as they could not be compared to “successful” runs. But for posterity’s sake (and for fun) we describe some of them here.

1. **Touch to Beam Hang** In this failure, the UI of the victim phone became stuck on the “Touch to beam” screen. Sometimes during the discovery cycle, the victim phone attempts to become the target and thus displays the Touch to Beam UI. In general, this UI disappears after a timeout if it is not used. It also goes away if the phones are physically separated. In the case of this failure, neither thing worked. The UI of the victim phone became totally unresponsive to input and the phone ultimately had to be restarted.

2. **Bad Battery Behavior** We discovered in the full drain experiments that often, an empty battery will not take a charge. Once it's been run into the ground, plugging the phone in is useless. The charging icon will appear frozen and the battery will not charge no matter how long it is left plugged in. The solution to this is to completely remove and reinsert the battery.
3. **NFC Subsystem Crash** The NFC subsystem has a tendency to crash occasionally. In fact, it happens often enough that by default, Android periodically checks to see if NFC is still functional and if not, attempts service recovery. Recovery is not always possible. We found that under certain scenarios, we could cause the NFC service to hang on the victim phone fairly frequently. In particular, running a 10 minute test with a fixed URI pointing to a 5MB file had a tendency to kill NFC on the victim phone.
4. **Too Many Notifications** On average, the automation framework sends 400 NDEF messages per minute. If all of those messages contain records pointing to downloadable URIs, the Android browser will attempt to download all 400 files at once. One of the results of this is that the Android notification screen will become overfull. There is a limit of 50 notifications per package in the notification screen, as shown in A.24. The notifications for downloads are all generated by the package *com.android.providers.download* and so the limit will be hit very quickly. Interestingly, the additional notifications are *not* discarded. They remain in the system, waiting to be posted. Once the first 50 notifications are cleared, the pending notifications will appear. The notifications persist even after restarting the phone. As a result, it is possible to have notifications appear for files that were downloaded several *days* prior. This quirk presents a potential sort of annoyance based attack against the victim phone. It could

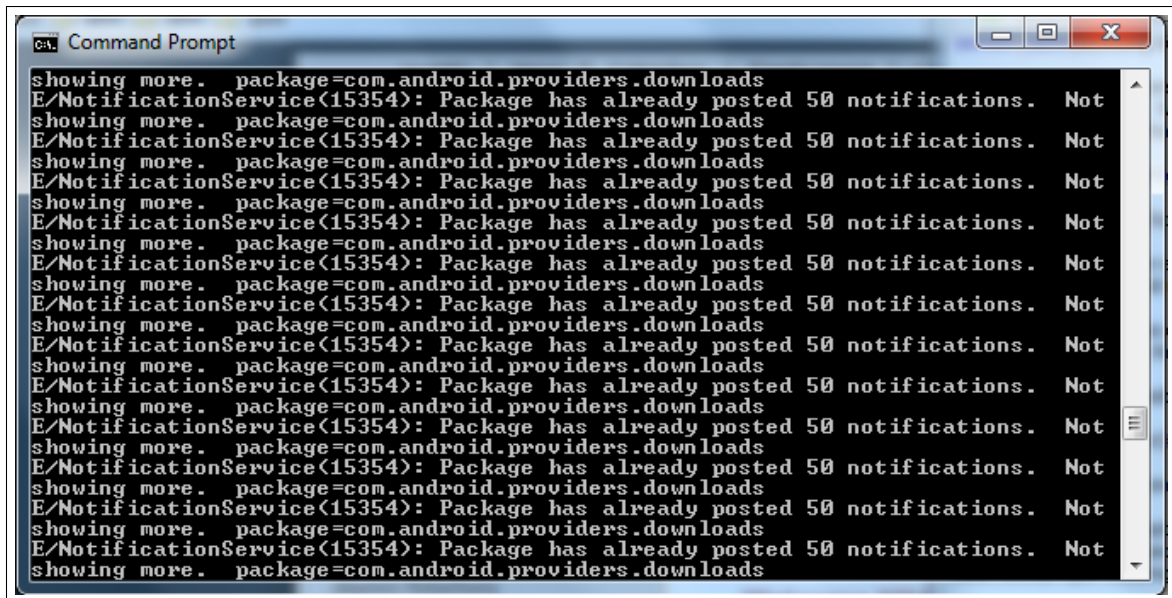


Figure A.24: Too many notifications

also be thought of as a denial of service in the sense that the user will lose legitimate notifications for purposefully downloaded files in the mess of attack notifications.

5. **Browser Hang** Again, we saw the most problems when we transferred a URI pointing to a larger file. One of the things that often occurred in this scenario was a browser hang as shown in A.25. In this case, the browser would become totally unresponsive. Sometimes it was impossible to kill the browser application and the phone needed to be restarted. Browser hangs become more frequent when the size of the file being downloaded is large; with a 5MB file, chances were high that a browser hang would occur during a ten minute test.

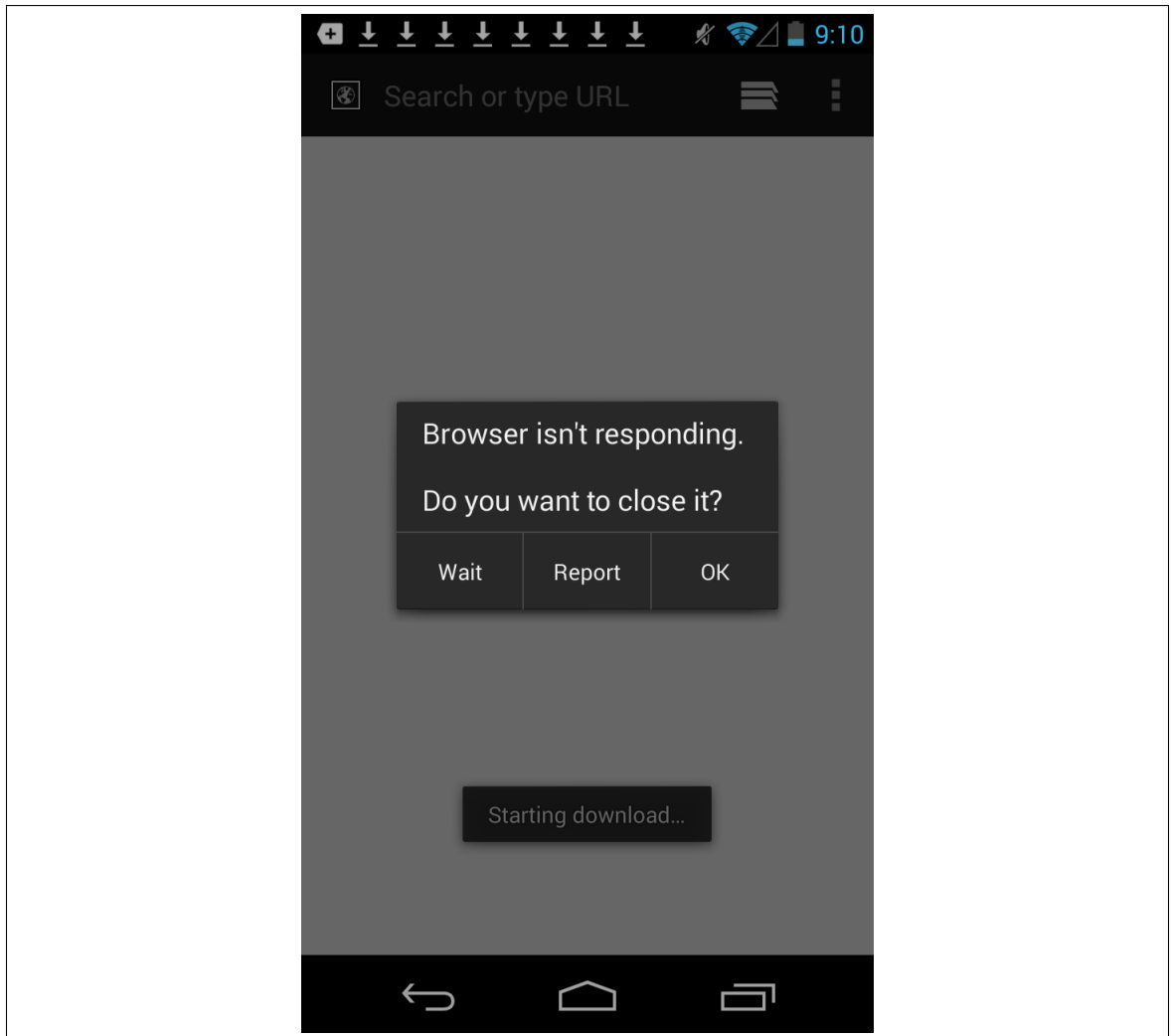


Figure A.25: Browser Hang

APPENDIX B

Additional graphs

In this appendix, we include some additional graphs describing our experiments.

B.1 Storage space change for various file sizes

Figure B.26 shows the individual experiments used to compute the average storage deltas for the 50kb file size experiments. Viewing them all side by side, it is easy to see that they follow roughly the same pattern. Though the tests were not done consecutively, the jump in available space centers very strongly around 6 minutes. It seems likely that the experiment itself is triggering the browser’s decision to empty the cache.

Figure B.27 shows the individual experiments used for the 500kb file size experiments. Again, we see roughly the same pattern across the experiments. For these experiments, the cache clearing happens a bit later, centered more around 8 minutes.

Figure B.28 shows the 5mb download scenarios. Here it becomes evident that most of the experiments are suffering from further complications on the victim phone. The included experiments all “appeared” to run. That is, there was no active browser hang or other phone crash. Yet clearly the experiments did not run consistently. Still, we can see in some of the cases that it follows the same established pattern as the smaller file sizes, just at a different scale.

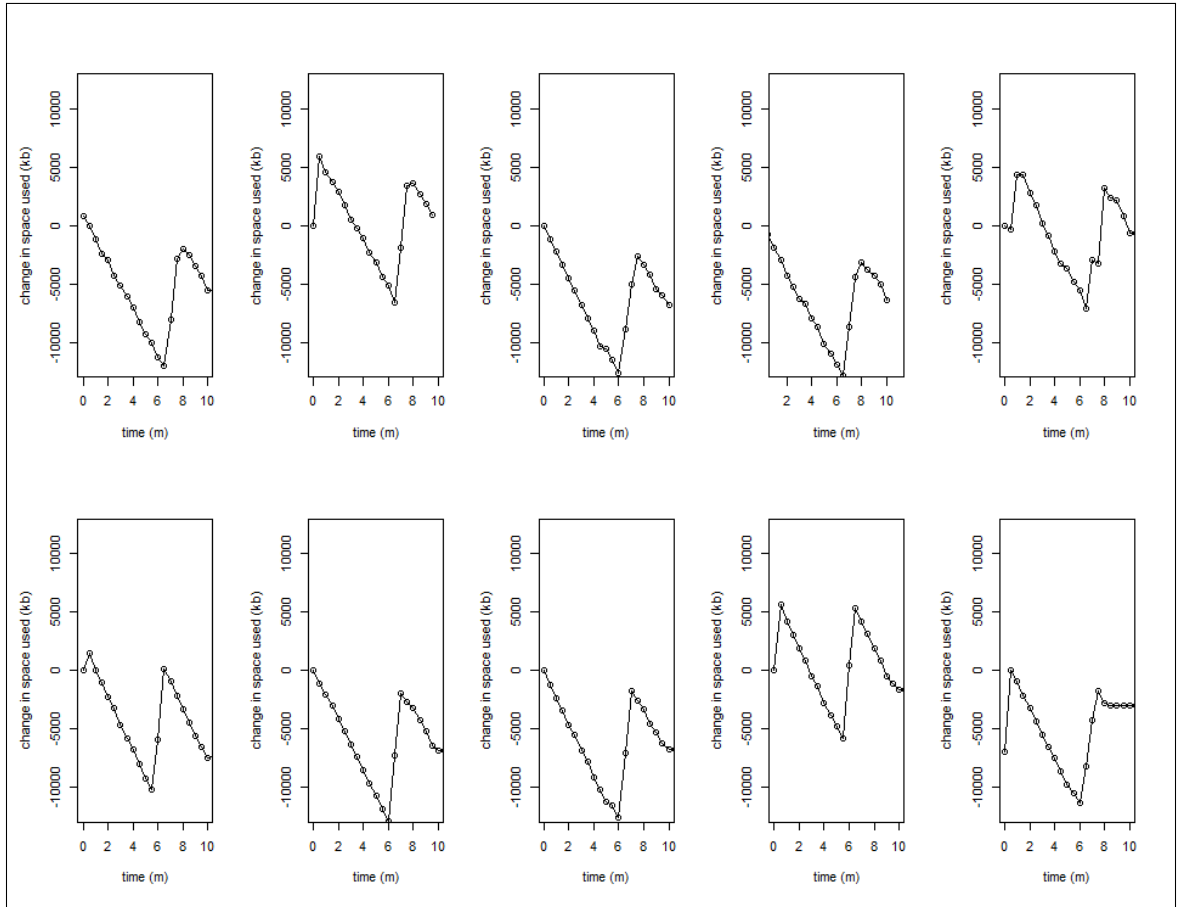


Figure B.26: Storage space change for file size=50kb

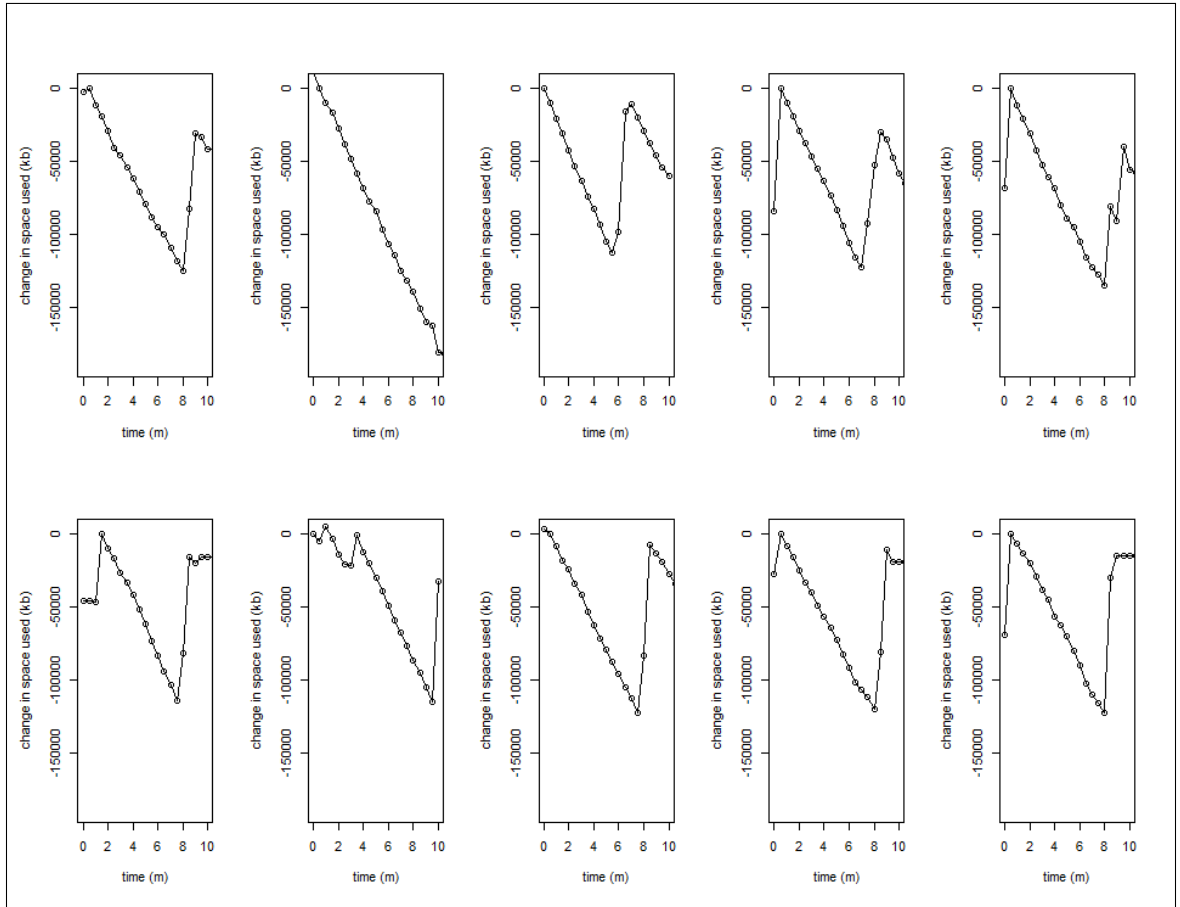


Figure B.27: Storage space change for file size=500kb

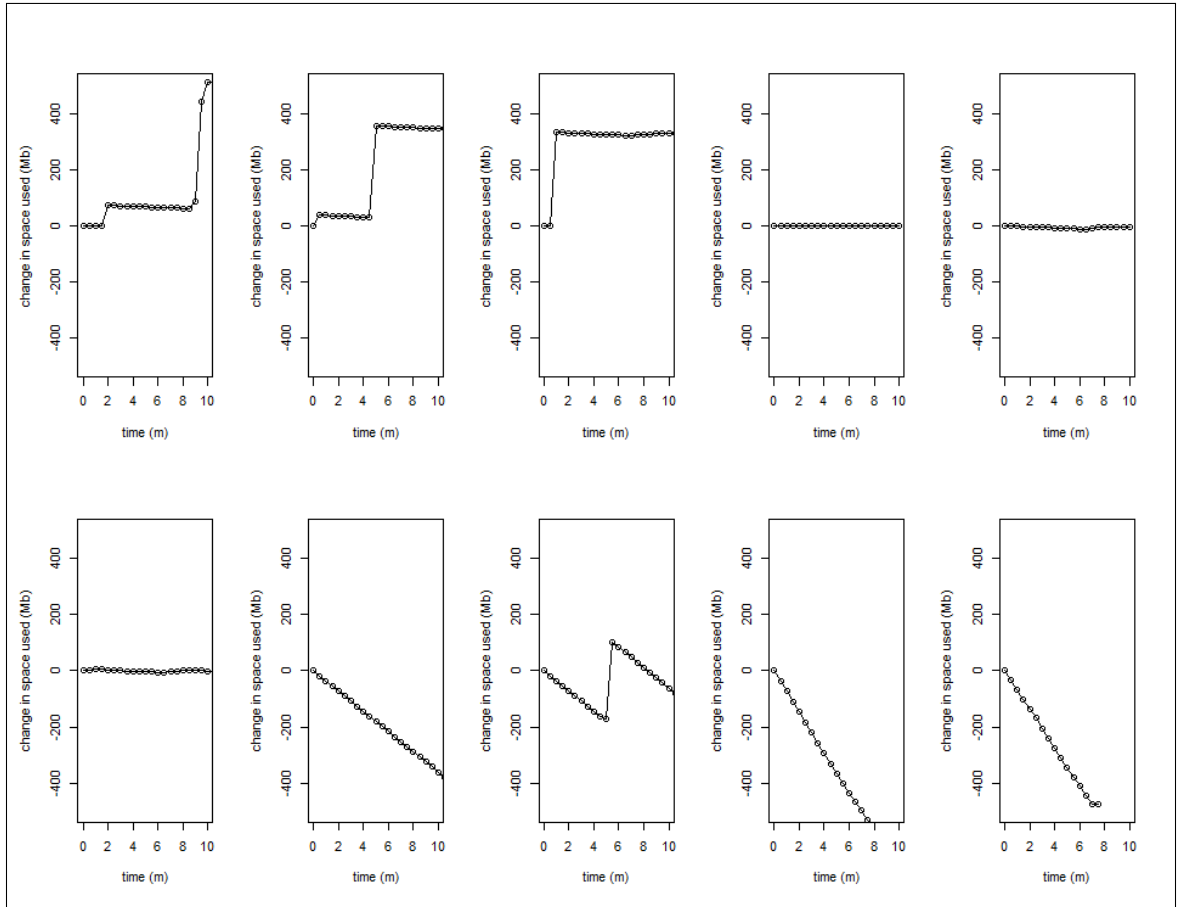


Figure B.28: Storage space change for file size=5mb