IMPERIAL COLLEGE LONDON

MASTER'S THESIS

MAJOR KEY

Author:
Preeya JOSHI

Supervisor:
Dr. Emil LUPU
Second Marker:
Anandha Gopalan

A thesis submitted in fulfillment of the requirements for a Master's in Computing

in the

Department of Computing

June 13, 2016

Abstract

When presented with a new application or internet connected device, in order to build an accurate threat model you first need to understand the protocol it uses. However, as encryption becomes more ubiquitous in the modern world, the protocols that applications use to communicate are often tunnelled within encrypted channels. If a security researcher wishes to analyse messages that an application receives, they must first bypass this layer of encryption.

There are existing tools that are capable of assisting a security researcher in obtaining cryptographic data from a running binary but whilst these tools are able to recover both the input and key they are unable to automatically differentiate between the two. Furthermore, the performance of these tools are far from optimal with large instrumentation and analysis overheads.

This thesis presents MajorKey , a cross platform suite of tools that automates the recovery of plaintext, key and ciphertext materials from applications using encrypted protocols. We utilise a novel method of distinguishing between input, output and key materials of cryptographic functions. This is achieved through performing statistical analysis on multiple run time traces of the same program with different inputs. Additionally, we present multiple performance enhancing techniques to produce significant speedups in the processing of application traces.

MajorKey is evaluated against seven different encryption algorithms including the AES competition finalist, RC6. The RC6 algorithm has been used in multiple real world contexts including malware distributed by the NSA and has not been analysed before in this context. Finally, we analyse MajorKey with a data communication application that receives encrypted data from the network via sockets. The results of this thesis show that it is possible to retrieve plaintexts and encryption keys in several use cases taken from real applications, in addition to demonstrating performance improvement of upto 6.5x over existing state of the art tools.

Acknowledgements

I would like to thank Dr. Emil Lupu, who came on board as my supervisor and saw potential in my project. Furthermore, thank you to Daniele Sgandurra and Federico Morini. The countless hours spent by them helping, proofing, critiquing and encouraging me to perform to my best ability has been truly appreciated.

Lastly, thank you to DJ Khaled for being DJ Khaled and motivation throughout.

Bless up 🗝

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis aims to automate recovering information from binaries handling data encoding through binary instrumentation and trace analysis. It aims to recover the inputs, including the cryptographic key and plaintext messages, and the outputs of the said program.

1.1 Motivation

Over the past two decades, technology has increasingly shaped our daily lives. According to the International Telecommunication Union, over 40% of the world now have access to Internet enabled devices [28]. A fact which is represented by a shift of human communication to media such as Facebook, Snapchat and Whatsapp.

For these applications to talk to each other across the Internet they need to agree to send information in a specified format and order, known as a *protocol*. Protocols are used to transmit data across networks, and between machines. Some are well-defined and open source: for these we are able to check that the protocols work as intended, and it easier for security researchers to discover vulnerabilities. Others may have incomplete or out of date documentation or are closed source making their assessment from a security standpoint harder.

Security researchers apply *protocol reverse engineering* techniques to both open and closed sourced protocols to gain a better understanding of how they work, and to ensure they are secure. They work by analysing the message structure that define how applications 'talk' to each other.

However, encrypting data is now more important than ever, as every information packet sent over the Internet can be read by any party that receives it whilst it's in transit. As a result, protocol encryption is becoming increasingly more common and due to it, intercepted network traffic is rendered as 'garbage' to prying eyes.

In order to continue analysing these protocols and applications to detect vulnerabilities, even when data is encrypted, we need the ability to read the plaintext messages being sent. The target application also requires the same information, and in order for it to correctly process the data it first must decrypt the ciphertext it receives. As we know the application will be able to decrypt the ciphertext it receives, a method for us to bypass encryption and recover the plaintext is to monitor the application as it runs. Then we can obtain the plaintext from the applications' memory when it has been decrypted and

reverse engineering methods can be applied as before on the plaintext. Furthermore, by monitoring the application as it runs means we can also recover the cryptographic key as it will be stored in a memory buffer that belongs to the application that will be read from during the run of the application. With knowledge of this not only will we able to decrypt data to understand the protocol being used or what messages are being sent, but be able to write messages using the key recovered.

Current methods of reverse engineering protocols are divided into two categories; network-based and host-based. Network-based methods sniff the traffic on the wire as it is in transit between two computers. However, as encrypted messages will appear as random data on the wire, it is not possible to use networked based methods to reverse engineer protocols with encryption. On the other hand, host-based protocol reverse engineering methods analyse application messages on a target machine. This allows for the memory holding the program data and the instructions run by the program to be accessed, something that is not possible with network-based reverse engineering. With encrypted data, however, they run in to a similar problem to that of network-based methods, as the incoming data to the application will appear as random.

This thesis monitors running applications to achieve its objectives and thus falls in to the category of host-based methods. It can be thought of a precursor step to protocol reverse engineering as it recovers the decrypted message that protocol reverse engineering methods work with. Once those plaintexts have been acquired, protocol reverse engineering can continue as before.

1.1.1 Other applications for MajorKey

Botnets

Botnets are an example of malicious software that use encrypted protocols that we want to analyse. Attackers have been known to infect computers with malware giving them remote control of a users machine without the user's knowledge. A networked group of these machines is known as a *botnet*.

Modern botnets have been seen communicating using undocumented encrypted protocols to communicate to their botnet masters [5]. This is a problem because an attacker can use these machines to illegally profit, cause harm to individuals and damage to companies. If we could break the encryption on the protocol, it would enable us to carry out protocol reverse engineering and communicate with the botnet. This will allow for better understanding of the protocol including how infected machines within the botnet communicate and how they exchange messages with the botnet master. With this one can disrupt their operations leading to taking down the botnet, re-purposing it or catching the owner.

Internet of Things

Today it is common for devices such as fridges, home central heating systems and televisions to be connect to the Internet. These devices reveal personal details about us, such

as when we wake up, watch TV or the kinds of foods we stock in our fridge. Ideally this information should remain private to us even though we have our devices connected to the Internet of Things, IoT, so we can turn up the heating as we head home from work.

With many of these Internet enabled IoT devices communicating over HTTPS, an encrypted protocol that sits on top of HTTP, using closed source protocols, it is harder for researchers to assess how secure they are, conversely how easy it is for an attacker to hijack the protocol and take ownership of a target's house.

Just like Botnets, these IoT devices will be running applications that receive encrypted data we send, such as a message to turn up the heating, and then they will decrypt the data so they can act accordingly. By monitoring the application as it decrypts the message we can recover the plain text and then apply standard protocol reverse engineering techniques to assess its security [13].

Ransomware

A further motivation arises from malware: most current malware implements advanced crypto-operations. Ransomware, for example, encrypts user files rendering them inaccessible until the owner pays a ransom for the decryption key. TorrentLocker [21] is a current iteration, first appearing in December 2014. It attacks a Windows user's files and demands a sum of one Bitcoin, currently £317, to decrypt them. The files are encrypted with the symmetric block cipher AES. The AES keys for the files are then encrypted with the asymmetric block cipher RSA making the files unrecoverable until the ransom has been paid. After that, the attackers provide the corresponding RSA private key to enable decryption of the AES symmetric key. If one had the ability to recover the AES key from the binary, one could bypass the need to pay the ransom as we would know the AES key and not need it to be decrypted.

To help address the aforementioned problems, this thesis aims to analyse programs encrypting/decrypting data and recover the inputs and outputs of the encoding operation.

1.2 Main Contributions

This thesis explores a novel method of recovering cryptographic materials from a target application, and it is evaluated against existing solutions.

Figure 1.1 gives a breakdown of the proposed approach. The figure is broken up to into 5 main steps.

To achieve step one, binary instrumentation, we first develop a tool to instrument a binary. Binary instrumentation allows us to analyse the behavior of the target binary through the injection of instrumentation code which allows us to monitor and record what the binary is doing. This provides us with the ability to generate a trace containing memory access details and the instructions called as shown in step two.

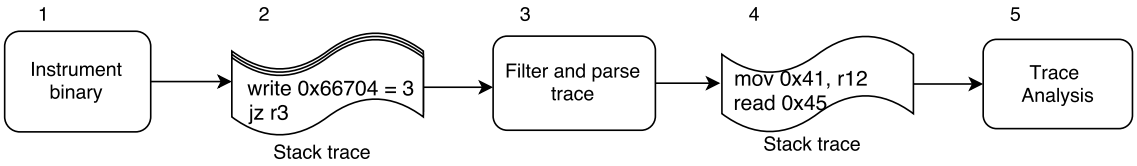

FIGURE 1.1: MajorKey : Solution Approach

The trace is then parsed, step three, by a tool that handles recovering cryptographic materials. Here we implement a parallelised parsing method to speed up the overall time required for analysis relative to current tool analysis times.

Then our method analyses the trace output to detect loops, step five. As cryptographic operations are known to occur in tight loops we employ loop filtration techniques such as detecting loops with a high number of bitwise arithmetic operations, common to cryptographic code instructions and not prevalent in general application instructions.

Furthermore, we also take a novel approach to recovering cryptographic material by statistically correlating multiple execution traces for the same the application to increase the result's confidence. This approach allows for increased confidence in the key being used as it will be the same across runs, and for inputs and outputs as one can verify relationships between the input, keys and output.

Overall, this thesis makes the following contributions:

- An architecture to recover cryptographic material from executable binaries.
- A Pintool to generate traces for target binaries.
- A Python tool that analyses said traces to recover cryptographic material.
- A new technique that aids in narrowing down potentially relevant cryptographic material.
- A parallelised approach speed up in trace parse and analysis time.
- The recovery of cryptographic material from applications carrying out XOR, DES, RC4 and RC6 operations.
- Recovery of cryptographic material for the RC6 encrpytion algorithm, which has not been done before.

1.3 Project Outline

This thesis presents its solution in 6 chapters.

The next chapter, Chapter 2, details related work in relative surrounding areas, alongside papers that have directly looked at the problem. Furthermore the background looks into

areas of interest that directly related to the goal of this thesis, such as instrumentation analysis and cryptographic algorithms.

Chapter 3 gives an outline of our proposed methods and it explains how key design details function.

In Chapter 4 the implementation of MajorKey is presented, and choice of tooling is analysed and discussed.

Chapter 5 provides an evaluation of our tool. The results of analysis with different programs are presented alongside an evaluation of our tool against other tools.

Finally in Chapter 6 we conclude the goals achieved and discuss future work.

Chapter 2

Background

As this thesis will produce a host-based tool, we begin by looking at strategies that enable instrumentation of the target application in Chapter 2.1. Within that chapter we cover the trace filtering method and taint analysis. To aid in key recovery, an understanding of how different cryptographic algorithms work is needed. Cryptographic algorithms fall into two main types, symmetric key and public-private, discussed in Sections 2.2.1 and 2.2.2 respectively. These are used to guide our loop filtering techniques in Section 3.6 to find cryptographic inputs and plaintexts. Finally, Section 2.3 looks at related works to this thesis, here we assess what they deliver, their strengths and where they have room for development.

2.1 Instrumentation

In order to recover cryptographic material used in a target application, we first must understand what the application is doing when it is executed. One method of achieving this is to monitor the running binary: this is done through a process known as *instrumentation*.

Instrumentation allows us to create a trace from the running binary. Most importantly, the trace it generates will contain the memory locations the application read from and wrote to, along with the values it read and wrote, and a list of every instruction called.

In our context, instrumentation is needed to recover cryptographic information as the application contains the necessary keys to decrypt the encrypted information being sent to the application. At some point whilst the application is running, the application will need to decrypt the data to process it. When it does, we can read the key in from program memory where the application will have stored it after decrypting it and use it to decrypt the input message and write the plaintext back to memory. All of this will be logged in the trace generated from instrumentation.

2.1.1 Dynamic vs Static Instrumentation

There are two different approaches to instrumenting a binary program, static instrumentation, and dynamic instrumentation. Static instrumentation happens before a program is run on the binary. An advantage of this is analysis is not dependent on the input and

all paths of the code can be checked, this provides full coverage. Static instrumentation rewrites a binary before it is executed and does not require another process at runtime to instrument the binary being executed. This means that the code can run faster as the only overhead will be from injected code. However the same level of detail dynamic analysis provides due to a lack of run time information that is provided in dynamic instrumentation.

During dynamic instrumentation, instrumentation occurs as the binary is being executed. This normally happens in a just in time compilation environment which achieves the goal by providing the developer with API methods to notify of when methods are being executed. These methods can then be probed at runtime during execution. However as the trace generated is now dependent on the run of an application and the inputs provided, full coverage of an application is not achieved. Furthermore, the need for an external process to be attached for run time instrumentation makes the original application slower. In order to determine what the cryptographic data during an applications run we need to see what data was read, written and instructions executed during the run of an application. To do this we need to use dynamic instrumentation. The rest of this thesis when talking about instrumentation will refer to dynamic instrumentation unless explicitly stated.

2.1.2 Taint Analysis

Instrumenting a binary usually generates large traces. As an example, some of the GPG encryption traces generated for the experimental evaluation were in the order of 1.8GB. Taint analysis is a method of reducing the trace file size, whilst ensuring that the trace will contain relevant information. It reduces the size of the trace, by instrumenting a program with respect to given input arguments, *taint sources*. Instead of outputting every instruction called, memory reads and writes, it only logs instructions, reads and writes that are related to those input arguments.

Initial taint sources can be user input, network traffic or user file. When instrumentation starts, the locations of the taint sources are marked. If an instruction is made using one of those marked locations, then taint will spread to the output of that instruction, *and any further instructions that use those outputs*. Similarly, if a tainted location is read from, the register that value is written to is further marked as tainted. On the other hand, when a tainted memory address or register is written to, then that location is removed from the set of tainted data.

```
1 READ    r1    [0x0000fejlkd] //read memory content into R1
2 ADD     r2    r1             //add R1 to the value in R2
3 MOV     r1    0x05           //move the value 0x05 to R1
```

LISTING 2.1: Taint Example

Listing 2.1 aids in the demonstration of the concept. In this example assuming memory address `0x0000fejlkd` was a taint source, the `READ` on line 1 will mark register `R1` as tainted. Then the `ADD` operation, line 2, will propagate the taint to `R2`. Finally, the `MOV` operation, line 3, will remove `R1` as a taint source and overwrite its value with `0x05`. At the end only `R2` will remain tainted.

A benefit of using taint analysis with instrumentation is a reduction in the size of the trace generated [25]. Through a reduction in trace size, parsing and analysing become faster. Furthermore, fewer false positives will be found when searching for potential cryptographic data as the trace will only contain reads and writes affected by the taint sources.

2.2 Common Cryptographic Algorithms

Cryptographic algorithms involve taking in a plaintext and a key, applying an encoding function using the key to the plaintext, and returning an encrypted message. Decrypting, is the opposite, involving an encrypted message, a key, and returning a plaintext.

The encoding and decoding function can often be the same, or one is easy to obtain if the other is held. Such ciphers are referred to as symmetric ciphers, discussed in Section 2.2.1. Asymmetric ciphers involve two different keys. The idea is to allow one key to be publically broadcasted so parties can encrypt messages using it, which only the holder of the corresponding private key can decrypt. With these ciphers one key can be used for encoding or signing and the other for decoding or verifying a message as discussed in Section 2.2.2.

This thesis will be looking at programs that receive encrypted data which they subsequently have to decrypt in order to process the messages they have just received. As this work aims to recover the key used by the cipher this section lightly touches upon common cryptographic algorithms that are used today. A better understanding of how different algorithms use keys will allow us to recover them more efficiently.

2.2.1 Symmetric Key Cryptography

Symmetric key algorithms use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext. The keys may be identical or there may be a simple transformation to go between the two keys.

An advantage of that is only one key needs to be remembered; however anyone who manages to obtain this key will have the ability to encrypt and decrypt data. The ability to encrypt data allows a third party to impersonate the person on the other end of communication. As a holder of the key can also decrypt data, they will be able to read any message sent, that was encrypted with that key.

One Time Pad

The simplest example of a symmetric key algorithm in the One Time Pad. The One Time Pad is the bitwise exclusive disjunction of a plaintext and a key. XOR works as shown in Table 2.1. If the two input bits are the same the output is a 1, otherwise the output is a 0.

TABLE 2.1: XOR Truth Table

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

When XOR is used in cryptographic operations, the key size is often the same as the plaintext, this is known as a *One Time Pad*. Detecting the key used in a one time pad is impossible, as it's an example of perfect encryption. However creating a true one time pad is hard because new key material for each message is required, and it has to be as long as each message. Therefore most algorithms 'cheat' and use a smaller input key to generate a pseudo random key of message length. These can be detect and is what is used in most encryption algorithms today.

DES

DES, the Data Encryption Standard, is a block cipher that was used in a wide variety of programs; today it is more of a legacy algorithm but still used [9].

Key facts about DES include:

- Fixed key size of 56 bits.
- Block size of 64 bits.
- 16 rounds per encryption.

These facts can be used to help narrow down our search space when searching for key material within an application using DES.

AES

DES was succeed by AES, the Advanced Encryption Standard [8]. It is also a block cipher that uses the same key for encryption and decryption. It supports three key sizes, 128, 192, and 256 bits, and uses a block size of 128 bits and usually involves applying 10, 12 or 14 rounds of the encryption function, depending on the key size used.

RC6

RC6 was one of five finalist for AES, the Advanced Encryption Standard and now is a proprietary algorithm patended by RSA Security [22]. RC6 supports three key sizes, 128,

192, and 256 bits, alongside that it used a block size of 128 bits. 10, 12 or 14 rounds of the encryption function are usually applied and this is dependant on the key size used with more rounds being used, in general, for smaller key sizes.

2.2.2 Public Private Key Cryptography

Public-key cryptography is a type of asymmetric cryptography. It involves two keys, a public key and a private key. The public key is usually widely disseminated and used to encrypt data that can only be read by the holder of the private key. Or, to verify a message that has been signed by the holder of the private key.

RSA

RSA is a widely use public-key algorithm used today [23]. Key sizes for the RSA private key and the public key modulus are usually of length 1024, 2048, or 4096. However today 1024 bit keys are not recommended as they are considered too short and easy to find with enough computing power.

Elliptic Curve Cryptography

Due to large RSA key sizes needed to keep RSA encryption secure, a current alternative is Elliptic Curve Cryptography.

ECC has the advantage of providing similar levels of security with smaller key sizes. An ECC algorithm such as ECDSA, Elliptic Curve Digital Signature Algorithm, with a key size of 256 bits should provide comparable security to a 3072 bit RSA public key [10].

The security of ECDSA, and ECC algorithms, comes from the algebraic structure of elliptic curves over finite fields. The the intractability of the problem makes it extremely hard for an attacker to recover an input without knowledge of the encryption key.

However, this same fact makes it easier to detect during instrumentation as an application using an ECC algorithm will have a high proportion of bitwise arithmetic operations when encryption/decryption is occurring. A feature that will be detectable in the trace.

2.3 Related Work

The related work has been split up into three main areas of interest. Section 2.3.1 looks at key works that attempt to recover decrypted ciphertexts. This is important as this thesis aims to do the same, the recovery of the plain text is the data that the application will act on. Section 2.3.2 touches on different approaches to classifying the encryption algorithm being used, and their strengths and weaknesses. Through knowledge of the algorithm being used, we can target our filtering methods accordingly to aid in recovery of the

cryptographic materials. Finally, Section 2.3.3 looks at current cutting edge techniques used to recover the encryption key being used.

2.3.1 Recovering Decrypted Ciphertexts

This section will look at various current works that look into decrypting cipher texts. All the approaches build on top of host-based automatic protocol reverse engineering techniques.

Automatically Decrypting Network Traffic

To the best of our knowledge, this is the first tool for recovering plain texts from an application that received encrypted input was proposed by Noe Lutz [18]. His thesis, "Towards Revealing Attackers Intent by Automatically Decrypting Network Traffic", proposed generic methods to locate when the input get decrypted and where in memory it is being stored. Lutz's solution parses the trace to extract details such as memory access patterns and the control flow, which show loops within the execution and function calls.

Then the extracted data is searched for features indicative of decryption. The thesis's proposed solution is to search for information entropy decreasing loops that use a relatively high number of integer arithmetic operations. As encryption functions are known to use tight arithmetic loops to increase security, a feature uncommon to normal program code, we can use that to enable plaintext recovery.

Lastly it uses taint analysis to follow instructions affected by the initial input e.g. encrypted messages to the application. An advantage of using taint analysis in his tool is it generates a smaller search space; however, if *under tainting* occurs, not all data related to the initial input will be caught, and vital buffers may not be marked as tainted causing his program to ignore it.

The tool was created using the instrumentation tool, Valgrind, making it native to Unix. Consequently, testing Windows malware is harder, as Wine is required and the malware may not run correctly. This thesis acknowledges this limitation and has taken it into account, see Section 3.3.2.

ReFormat

ReFormat [31] expands on a previous tool by the same group called AutoFormat [16], to allow for automatic reverse engineering of encrypted messages. Their approach follows assumptions made by Lutz and this thesis in that an encrypted message will go through a decryption phase and then normal protocol processing. Furthermore, that the two phases will be differentiable as the decryption phase will use a significantly larger number of arithmetic and bitwise operations.

ReFormat first identifies the decryption phase, and the transition point between it and normal protocol processing. This is achieved through analysing with respect to a set threshold, the cumulative percentage of arithmetic and bitwise instructions. The transition point is declared as the point between the instruction with the maximum cumulative percentage and the one with the minimum.

Similar to Lutzs' work, taint analysis is used to reduce the search space of potential buffers. They use taint analysis differently though with Reformat marking all buffers at the beginning of the decryption phase as tainted while Lutz only considers those directly affected by encrypted inputs. As the two tools have been tested on different protocols it is hard to directly compare the advantages of the different approaches. Lutz successfully recovers plaintexts for OpenSSL AES, Blowfish and Twofish whilst ReFormat focuses its evaluation on protocols and recover data for HTTPS, IRC, MIME and an unknown protocol used by the malware Agobot.

Lutz's work and ReFormat report that the decryption phase and the normal protocol processing phase will be differentiable and use it as a foundation for building their tools.

Dispatcher

Bitblaze released Dispatcher, a tool that differs from the aforementioned as it is not limited to protocols where there is a boundary between decryption and protocol processing [6].

Dispatcher is interesting to this thesis as it does not rely on taint analysis. Instead, it runs over the execution trace and calculates the ratio between arithmetic and bitwise operations for each function in the trace. It identifies any function above a certain threshold as an encoding function. Then it calculates the buffers holding unencrypted data by computing the write set for the decryption routine.

Dispatcher combines Bitblaze's method of identifying cryptographic thresholds with ReFormat's notion of a phase difference between decryption and normal protocol processing to locate the outputs of decryption and, in turn cryptographic loops.

CipherXRay

CipherXRay is the most recent work that looks into decrypting messages [15]. The main aims of CipherXRay is to recover encrypted data, identify the cryptographic operations used and recover all transient secrets such as keys: the latter parts are discussed in Sections 2.3.2 and 2.3.3 respectively.

Unlike previous works, CipherXRay takes a novel approach and instead of instruction profiling the execution trace it searches for evidence of the avalanche effect [32]. The avalanche effect of cryptographic functions states that a one bit change in the input results should result in an over 50% change in the output ciphertext.

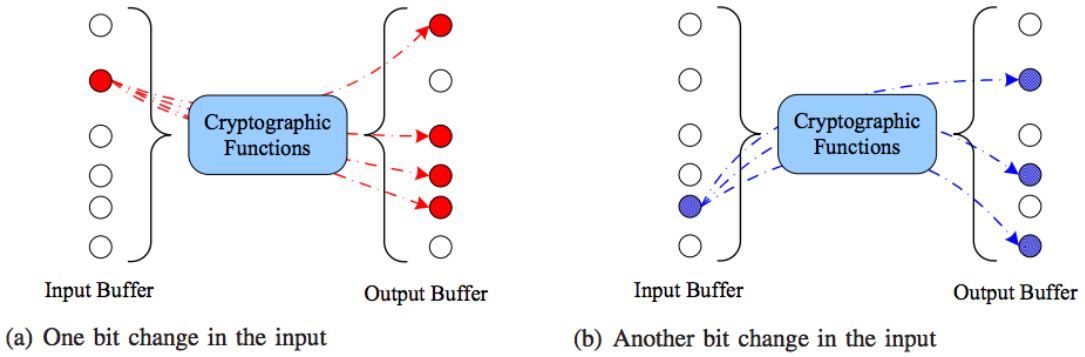

FIGURE 2.1: The Avalanche Effect [15]

CipherXRay first uses taint analysis to narrow down candidate buffers and then searches for the avalanche effects between two sets of contiguous buffers. It encodes the avalanche effect as shown in (a) and (b) in Figure 2.1. The one bit input change goes on to cause 50% of the output bits to change. Using this they show that if a byte were in the encrypted input it's effect would be seen in nearly all the output. Once the avalanche effect has been detected, the decrypted data can be retrieved from the buffers at the other end of the avalanche effect.

CipherXRay has the advantage of working on applications with multiple rounds of encryption like the previous tool, Dispatcher. Furthermore, it has the potential to work on obfuscated binaries and was successfully tested on a packed version of Stuxnet [14].

CipherXRay successfully recovers cryptographic data for Blowfish, AES and SHA-1 using the avalanche effect. This thesis differs from CipherXRay as it does not use the avalanche effect to locate inputs and outputs as the avalanche effect has weaknesses when analysing stream ciphers. Each input data bit to a stream cipher only affects one output data bit, therefore does not produce the avalanche effect. As a result we build upon aforementioned tools and uses a novel technique involving multiple execution traces from the same program, discussed further in Chapter 3.6.2.

2.3.2 Classifying an Encryption Algorithm

The classification of encryption algorithms has importance in the field of security, as knowledge of the type of algorithm being used allows one to find exploits for said algorithm. Through this, an adversary can either listen in to communication or go further and impersonate a party involved in communication or forge a document.

We will look at various signature-based methods that currently exist and analyse the weaknesses and strengths of the approaches. It is useful to this work as different cryptographic operations use keys in different ways, and knowledge of which operation is being used will help us locate the key being used.

CipherXRay

As mentioned above, CipherXRay uses the avalanche effect to find plaintexts. Similarly to classify different types of encryptions, it looks at the signature exhibited by the avalanche effect, as it is manifested differently by different algorithms.

An example of one of heuristics it uses to detect encryption methods is its hash detection. If the output of an encoding operation is smaller than the input then it knows operation is not a cipher but in fact a hash. Furthermore, if the encoding operation output is of a fixed size for inputs of different sizes then it's probably a hash. It then confirms this using the avalanche effect, as any input byte will affect all output bytes.

A fallback of the tool is its inability to detect stream ciphers. This is as stream ciphers will not show an avalanche effect because any byte of the stream will only affect one corresponding byte in the output. This thesis has taken this in to account and opted for a strategy that will not prevent stream ciphers from being detected. This is demonstrated in our Chapter 5.3 where we evaluate our tool on an XOR cipher.

Identification of Cryptographic Primitives in Binaries

Grobert et al's approach also identifies cryptographic primitives using dynamic analysis [12]. This means that they analyse the application in question by generating a trace of the application executing actual data, in this case the encrypted text being passed into the program. In their paper they present several techniques which we will evaluate.

Their work builds on top of observations detected by Lutz on unique attributes of cryptographic functions, mentioned in Section 2.3.1. Using that as foundation they propose the following heuristic based identification methods:

- Chains heuristic which instruction sequences are compared against a set of existing patterns. This involves using a data set containing known signatures and patterns from encryption algorithms, and then looking for those patterns in the trace being analysed.
- Mnemonic-Const heuristic which extends Chains and pattern matches on instructions and static constants. As different cryptographic methods use unique constants, e.g. every MD5 implementation they tested contained ROL 0x7 and ROL 0xC, testing for these *signature* instructions is a good indicator of the instruction being part of an encryption function, if they are found.
- Verifier heuristic which identifies relationships between the input and output of a cryptographic loop. With knowledge of the plaintext, key and candidate ciphertexts, the inputs are run through a reference implementation of the algorithm to see if they match. This provides the assertion that the correct data has been selected, and allows for selection of the correct data if multiple potential inputs and outputs have been detected.

We note the main limitation of pattern-based methods, including the ones from Grobert et al.'s paper is if the cryptographic algorithm is not public, and in the pattern database then it will not be identifiable. Furthermore unlike CipherXRay, it is unable to recognize algorithms that used multiple different encryption operations.

Aligot

Aligot looks at identifying cryptographic operations in obfuscated programs [7]. Although this work will not look at obfuscated programs the approaches taken are novel and of interest to this thesis. Simply Aligot compares the I/O parameters of operations to known cryptographic functions to determine what algorithm is being used. The authors observe that a single I/O pair is enough to identify most cryptographic operations.

It then uses dynamic analysis with control flow analysis to identify cryptographic loops and its input and outputs. It then attempts to find a loop in the original program that returns the same output as a reference algorithm, in their database, for some input. If they find a match, then they can successfully classify the algorithm. The authors noticed that the longest task in their analysis was loop detection, and it was not relative to the execution trace size. Recognising this as a disadvantage, our work has taken advantage of spacial locality of instructions and has parallelised the loop detection process as discussed in Section 4.5.2.

2.3.3 Recovering the Key

The task of recovering the key used in cryptographic operations has been mainly looked at a static analysis problem. These tools require keys to be stored in plaintext form in a binary, and are less effective against applications that encrypt keys or generate them on the fly.

In this section, we will discuss exsisting static analysis methods, and evaluate their strengths and weaknesses. We will also look at how CipherXRay retrieves keys, as it approaches the problem with dynamic analysis and can recover encrypted and transient keys.

Our research has shown that CipherXRay is the most versatile at recovering the keys. This is due to it being a generic algorithm as opposed to being tied to specific cryptographic implementations. However it cannot identify keys that are not stored in contiguous buffers.

Playing Hide and Seek with Stored Keys

Static approaches to finding keys have been looked at in various contexts. Shamir and Someran try to identify cryptographic keys in large amounts of other data [26]. Although this is not directly related to this work, as they look at static binaries, we also play hide and seek for the key under mounds of other accessed data.

One method relevant to this work is identification of high entropy region. Their paper does this through visual analysis, whereas if we were to adopt this approach we would encode the definition of a high entropy region look for that. Looking for a high entropy region works as cryptographic keys are random strings that often contain more information than the average plaintext. This method only works in the context of a binary not being obfuscated, as obfuscation will increase entropy. With an increase in average entropy, keys are less likely to stand out and will be harder to recognise.

CipherXRay

Once the input, output buffers and the cryptographic operation have been identified, the first task involves identifying if a key was used. It identifies keys as they will exhibit the avalanche effect, discussed in Section 2.3.1, on the output of the function. However the main task faced is isolating the key from other sources that also exhibit the avalanche effect on the data.

Other sources include initialisation vectors for the cryptographic operations, static data used by operations, e.g. S-Box in AES and intermediate results that are not the output data. To find the key used, and isolate it from potential noise sources, it combines the avalanche effect pattern with data liveliness analysis. As the key, or a derivative of the key, is needed during every round of encryption, data liveliness analysis will detect the key or round key's usage. This will allow us to differentiate between it from other sources such as the initialisation vector which will only exhibit the avalanche effect on the data in the first round of encryption.

Using data liveliness it can distinguish the key from initialisation vector, or other data inputs to cryptographic operations as the data will differ every block, whereas the key will be used on every block.

It differentiates the key from Staticc data used in cryptographic operations using the knowledge that a key would affect the whole output, whereas static data usually impacts a small amount. This approach is different to other methods which identify similar sources using signature detection. As they often involve constants that are key to the cryptographic functions as is the case with S-Boxes.

Overall CipherXRay's approach to key finding provides a smart insight in that the same key will be used in multiple rounds of a cryptographic operation making it easier to identify.

2.3.4 Summary

To summarise, our main related tools work as follow:

- **CipherXRay**: Uses the avalanche affect to detect the difference between the input and the output of a loop.
- **Lutz**: Identifies cryptopgraphic loops those with decreasing entropy.

- **ReFormat**: Uses cumulative bitwise and arithmetic operations to detect the boundary between encryption and normal protocol processing.
- **Dispatcher**: Identifies encoding functions as those containing more than a threshold number of bitwise and arithmetic operations

All of the above tools are able to recover cryptographic keys, inputs and outputs. The advantage of CipherXRay is it's highly accurate for block ciphers but does not work with stream ciphers as the avalanche effect cannot be detected. Lutz method does not work with obfuscated binaries and is not cross-platform which means detecting cryptographic data on Windows binaries is harder as a Windows emulator is required. An advantage of ReFormat is that it has been tested and proven to work with encrypted protocols, E.G. `HTTPS` whereas the others have been mainly tested with encryption algorithms in simplistic applications and one larger case study.

These works lack in failing to identify the difference between the key and the input passed into the encryption algorithm. When a key is used multiple times, being able to differentiate between the two can aid in reovering futher pairs of inputs and outputs as the key will exhibit the same effect on different pairs. As previous works do not analyse more than one input to an application they cannot use multiple traces to increase the confidence of their results, or distinguish the key from the input.

Chapter 3

Design and Architecture

3.1 Approach and Assumptions

Our goal is to recover information not just for a specific application or encryption algorithm, but for a wide range of applications implementing cryptographic algorithms. More specifically, we want to recover plaintexts for any encrypted message sent to an application using XOR, DES, AES, RC4 and RC6 encryption. Furthermore, we would like to recover the decryption key used by the application to recover the information.

In doing so we make certain assumptions about the target binaries so we can direct our filtering and analysis accordingly.

The first, as observed by Lutz, is that decryption operations are likely to occur in tight loops [18]. An example of this is block ciphers, discussed in Section 2.2.1, where there is a loop that runs over the input buffer to decrypt the input block by block.

Secondly, in order to recover the keys and inputs, this thesis looks for the high ratio of arithmetic loops and as such we assume there is a quantitative difference between the decryption phase and the normal protocol processing phase. This holds for cryptographically secure algorithms as their security is derived through number theory which involves a combination of repeated applications of bitwise and arithmetic operations.

Finally, we assume that we can run the application in question and that it has not been obfuscated or packed. A key assumption made is that any inputs and the output of the cryptographic function each reside in continuous memory buffers.

A summary of key assumptions and observations made:

- Cryptographic operations are highly likely to occur in loops.
- The decryption process contains considerably more arithmetic and bitwise operations than normal protocol processing.
- The application being monitored has not been obfuscated to avoid dynamic analysis.

3.2 Design Overview

MajorKey is an end-to-end suite of tools to provide the ability to recover cryptographic data from programs that encrypt messages. The problem can be broken up into the following flow as shown in Figure 3.1.

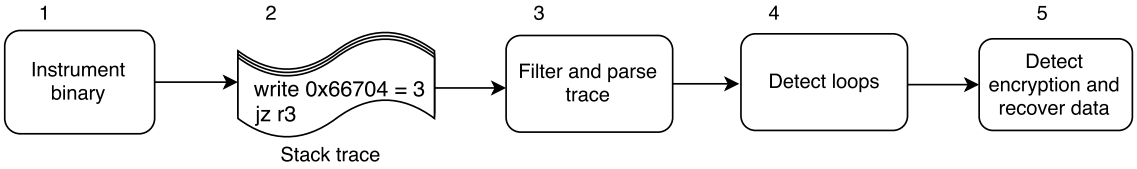

FIGURE 3.1: MajorKey : Approach

As shown at point one in Figure 3.1 the binary is first instrumented. Instrumenting the binary will allow us to monitor the item being instrumented as it runs and generate an instruction trace. The binary passed into the instrumentation tool will be the application of interest that carries out the encryption we are trying to detect.

The analysis part of our tool can accept traces generated by multiple software. We choose to use Pin [3] to instrument binaries as it is a cross platform framework. We discuss this further in Section 3.3. Once the target binary has been instrumented, an instruction trace is produced to produce containing information about memory reads and writes, and instructions called by the program. This is depicted at point two in the figure and is explained further in Section 3.4.

The trace is then filtered and parsed, point three, into an observable, analysable format. This involves reading the trace into memory and structuring the data so it can be analysed. The trace is then processed in step four to detect potential loops containing cryptographic operations, this is expanded upon in Section 3.5. Finally in step five the loops are analysed and filtered, to detect and recover the cryptographic material. Such filtering processes include looking for loops with a high number of bitwise arithmetic operations, e.g xor usually found in cryptographic functions. Our methods are further explained in Section 3.6.

3.3 Generating Instrumentation Trace

There are already many existing binary instrumentation frameworks to analyse application results. Our tool is capable of working with any of them, as long as the trace generated by that framework follows the format described in 3.4. MajorKey has been tested with applications instrumented using the dynamic binary instrumentation framework, Pin [3].

3.3.1 PIN

Pin is a dynamic binary instrumentation framework. It instruments code in a *just-in-time* fashion as it runs without the need to recompile it. We chose to use Pin due to it being multi-platform: it is able to run on Windows, Linux and Mac OS. Furthermore it can be used to instrument all the user level code of an application.

Using the Pin API, one can create *Pintools* that allow a greater understanding of the application being instrumented. Some of its relevant capabilities to MajorKey are:

- Replace application functions: this is advantageous because if the program makes a call to a malicious function, which may cause harm to the machine running the Pintool, one can skip over it.
- Examine every application instruction executed by the program
- Insert your own functions that can be called each time an instruction is executed or a specific type of instruction is called.
- Track function calls made by the application, and examine arguments, such as values being written to memory addresses.

3.3.2 Reducing Trace Size

Instrumenting an application can often produce very large output traces. As traces grow in size, the time required to parse and analyse them increases and as such our tool takes longer to run overall. An example would be traces generated by GnuPG applications, whose traces are over 1.8 GB. This section explains the methods used to reduce the trace size generated by our tool without compromising the quality of our results.

We apply a preprocessing method that identifies regions of memory that contain relevant instructions and creates a reduced trace by only including instructions from that region. In the traces we generate, each instruction is located at a point in memory. On a 32 bit computer architecture instructions are usually 32 bits apart. This is important to our thesis as, if we can analyse a binary beforehand and detect memory regions of interest, we can filter out just those instructions thus reducing the size of the trace.

```
1 0x00007f83468cd1f8    ret
2 0x00007f83468cd1fb    lea rdx, ptr [rip+0xf24f]
3 0x00007f83468cd1fe    mov rsp, r13
4 0x00007f83468cd22c    jmp r12
5 # A instruction call was made here, but not logged by our trace
6 0x0000000000400e3a    xor ebp, ebp
7 0x0000000000400e3c    mov r9, rdx
```

LISTING 3.1: Example Instructions with Addresses

Listing 3.1 shows a typical example of what we can expect to see when running a program. The first argument in each line, e.g. `0x00007f83468cd1f8`, is the address of the instruction being executed. The arguments that follow are instruction and its arguments.

Each address is given in hexadecimal, and as can be seen each instruction is 4 bytes after the next, or 32 bits.

By looking at the last two instructions after the comment, we can see that they reside in a different area of memory. To understand memory addresses used in a program, we can analyse the memory used by our program further by using a tool called `pmap` on our running application. `pmap` provides a memory map of the process running our application and allows us to see what areas of memory different parts of our application use.

```
$:  ./home/derp/examples/a.out & pmap $!
    0000000000400000      4K r-x-- /home/derp/examples/a.out
    0000000000600000      4K r---- /home/derp/examples/a.out
    0000000000601000      4K rw--- /home/derp/examples/a.out
    00007f2a570f2000   1772K r-x-- libc-2.19.so
    00007f2a572ad000   2044K ----- libc-2.19.so
    00007f2a574ac000     16K r---- libc-2.19.so
    00007f2a574b0000      8K rw--- libc-2.19.so
    00007f2a574b2000     20K rw---   [ anon ]
    00007f2a574b7000    140K r-x-- ld-2.19.so
    00007f2a576c3000     12K rw---   [ anon ]
    00007f2a576d6000     12K rw---   [ anon ]
    00007f2a576d9000      4K r---- ld-2.19.so
```

LISTING 3.2: Pmap Output

As can be seen in Listing 3.2 output all code from our application, `a.out`, is generated in the memory region `0000000000400000 - 0000000000605000`. This knowledge can be obtained by running the command `'APPLICATION' & pmap $!` from the command prompt for the application in question. With this knowledge we can then instruct our instrumentation tool to only look at instructions from certain memory regions, or conversely to ignore specific regions we know are not important for our analysis. This greatly reduces the time spent in analysing the trace.

Furthermore access to the binary means we can analyse it to build up prior knowledge of what libraries and functions are called. We can use the knowledge we build up of libraries called to exclude instructions that are called from system libraries that do not execute cryptographic code. This method is not used to whitelist functions we analyse as we cannot guarantee that all of the functions necessary for encryption will be included, whereas blacklisting common libraries such as those that handle I/O will produce no false positives.

Depending on the binary in question, this can be obtained through using a simple program such as `strings` which prints all consecutive printable character sequences greater than four characters long, or using with a more complex tool such as `objdump` which displays information about object files.

```
vagrant@derp:~/examples$ objdump -t DES/run_des.o

DES/run_des.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000400238 l    d  .interp      .interp
* 0000000000603c40 g     O .data      initial_message_permutation
* 0000000000603b20 g     O .data      sub_key_permutation
0000000000000000       F *UND*        fwrite@@GLIBC_2.2.5
0000000000603e20 g     O .data        .hidden __TMC_END__
```

```
0000000000000000  w      *UND*          _ITM_registerTMCloneTable
```

LISTING 3.3: Objdump Output for DES

Listing 3.3 is a snippet of the output from running `objdump` on our DES program. The full output can be found in **AppendixA**, it shows a few entries from the symbol table of the program. From reasoning about the output, we can conclude that instructions relating to the two lines starred will probably be of interest due to the names associated with the memory addresses, mainly `initial_message_permutation` and `sub_key_permutation`, both of which are related to cryptographic functions. The above technique is useful in reducing the trace time and file size. Furthermore, it can allow us to detect functions we may want to skip in the code such as an unpacker. However, although out of the scope of this thesis, it is worth noting that a program may be obfuscated to prevent such analysis by scrambling the symbol names.

3.4 Instruction Trace Format

As mentioned previously, regardless of the instrumentation tool used to generate the trace for a given application, our analysis tool will be able to detect the cryptographic data within the input. To do this we have a standardized format that we expect the trace passed into our analysis tool to follow. Each line in the output trace can be one of three types, a memory read, a memory write or an instruction. They have the respective formats:

- A memory read:
  `r | size_of_memory_to_be_read | value_read = memory_address_read_from`
- A memory write:
  `w | size_of_memory_to_be_written_to | address = value`
- An instruction:
  `address | img | routine | offset | disasm | extra_data`

A `r`, `w` are tokens we use to distinguish between memory reads and writes respectively and the `size_of_memory_*` field signifies whether it's a 8 bit, 32 bit or 64 bit addresses accessed. For the instruction format, the `img` and `routine` fields signify the module and function the instruction comes from and the `offset` is the offset of that instruction from the starting instruction for the given function. The `disasm` is the machine level instruction that will be executed and `extra_data` contains any necessary data needed for the machine level instruction to execute.

```
0x00007fe6284a3a96|/lib64/ld-linux-x86-64.so.2|.text|0x000000003fb6|
        mov qword ptr [rip+0x21e1fb], rdx
w  64 0x7fe6286c1c98 = 0x52802720515
0x00007fe6284a3a9d|/lib64/ld-linux-x86-64.so.2|.text|0x000000003fbd|
        lea rdx, ptr [rip+0x21e3cc] | rdx = 0x7fe6286c1e70
0x00007fe6284a3aa4|/lib64/ld-linux-x86-64.so.2|.text|0x000000003fc4|
        mov r13, rdx | r13 = 0x7fe6286c1e70
r  64 0x222e70 = 0x7fe6286c2000
```

```
0x00007fe6284a3aa7|/lib64/ld-linux-x86-64.so.2|.text|0x000000003fc7|
        sub r13, qword ptr [rip+0x21e552] | r13 = 0x7fe62849f000,
        rflags = 0x206
```

LISTING 3.4: Example Trace Output

Listing 3.4 gives an example of a trace output containing all three types. The token "|" has been used as a delimiter as it will never naturally occur in any value in the trace output and our program can therefore reliably split using it. As can be seen above there is one write, one read and four instructions in the trace output. The instructions are all from the `/lib64/ld-linux-x86-64` shared object library and from the function `.text`.

The `disasm` field is further broken up to `mnemonic | op1 | op2`, and for different architectures this work also expects a mapping from `mnemonic` to `instruction_type` to be passed. This provides semantic information about the instruction being executed such as if it's an arithmetic operation, a branch or a logical instruction. Such information can be found in the instruction manual for the given architecture in question. This work looked at the `X86` instruction set manual to generate the above types [17].

3.5 Detecting Loops in Trace

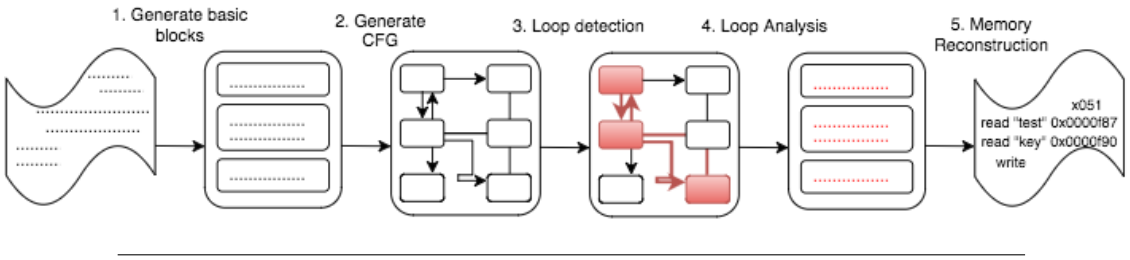

FIGURE 3.2: Trace Analysis Breakdown

Detecting loops from an input trace involves the following steps, and has been illustrated in Figure 3.2:

1. Detect basic blocks in instruction trace.
2. Generate a control flow graph from basic blocks.
3. Detect loops from basic blocks and control flow graph.
4. Cryptographic loop analysis.

From past works [7], discussed in Chapter 2, it has been established that cryptographic operations normally take place in tight loops, containing few instructions and a large number of iterations. Therefore the first three stages prepare the data for the fourth. These loops can then be passed to our filters as discussed in Section 3.6.

3.5.1 Detecting Basic Blocks

Basic blocks are generated from the output instrumentation trace to reduce the number of objects we have to work with. A *basic block* is a group of instructions that form a *straight-line* sequence with no branches except the entry point and the exit point. It reduces the size of the trace by grouping instructions together thus making further analysis and loop detection easier.

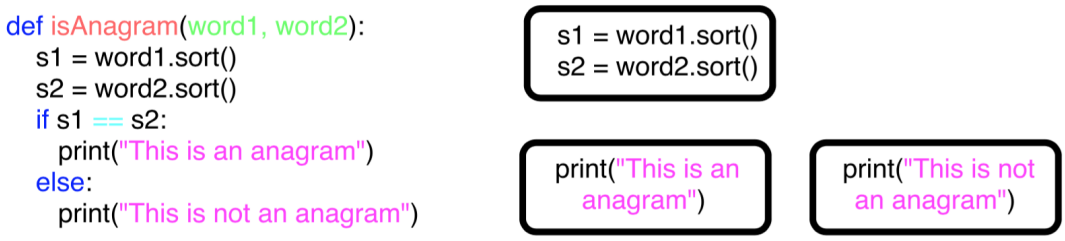

FIGURE 3.3: Basic Blocks Example

Figure 3.3 shows a simple Python function that can detect if two words are anagrams of each other. The right side of the diagram shows the basic blocks it would be broken up to. As can be seen for each basic block there is one entry point, the start, if the first instruction is executed then we can guarantee they all are. We use this fact to reason about programs at a higher level, that of the basic blocks, and in turn it make our program faster and more efficient.

We generate basic blocks directly from the instrumentation trace: this means we can guarantee that the basic blocks are an exact representation of the application run as every instruction within the trace will appear in a basic block. We use the algorithm shown in Listing 3.5 to detect them.

```
1  def generate_bbs(instructions):
2      bbs = []
3      prev_branch = True
4      block = False
5      for i in instructions:
6          if prev_branch:
7              if block:
8                  bbs.append(block)
9              block = BB(i)
10         else:
11             block.append(i)
12
13         if i.is_branch():
14             prev_branch = True
15         else:
16             prev_branch = False
17
18     return bbs
```

LISTING 3.5: Basic Block Detection Algorithm

In the first part of the algorithm, the instructions are run through in the order they are executed and appended to a basic block. If the instruction is a `branch` instruction, causing

a different instruction sequence to execute and it to deviate from the next chronological instruction, then the variable `prev_branch` is set. On the next execution, if the previous instruction was a branch instruction then the current basic block is finished, appended to the list of basic blocks and a new one is started.

We use the method `is_branch` defined on an `Instruction` object to determine whether an instruction is a branch or not. This is done by first getting the instruction mnemonic of the instruction being executed from the disassembly of the instruction object. The mnemonic is then looked up in a type table generated from the X86 Opcode and Instruction Reference [17] to see if the given type is a `branch` instruction.

TABLE 3.1: Size Comparison between Instructions and Basic Blocks

Target program	Instruction Count	Basic Block Count	Percentage Reduction
XOR	154532	21788	85%
AES	19695088	195009	99%
DES	660759	48610	92%
GnuPG	20887537	442326	97%

Table 3.1 gives an example of size reductions seen by various inputs after generating basic blocks from trace instructions. As can be seen using basic blocks provides on average over a 90% reduction in number of objects we have to analyse, with slightly less for the XOR application but that is likely due to the application being very small to start with.

To summarise, we build up basic blocks from the instrumentation trace in order to reduce the number of unique objects our analysis tool has to handle. We then use the basic block representation of the application to generate a control flow graph, this is discussed in the next section.

3.5.2 Control Flow Graph

To understand where decryption is occurring we need an accurate representation of the target application so we can detect loops. This work uses a control flow graph, CFG, to represent the path taken by the target application during its run. Basic blocks already reduce each individual instruction in our trace to blocks of instruction, and a CFG allows us to understand how these blocks interact.

Our CFG is a directed graph where the vertices represent basic blocks, and the edges show the control flow between them. The edges are representatives of `jumps` and `calls` within the program. An advantage of using basic blocks rather than individual instructions at our vertices is that the overall CFG will be considerably smaller whilst providing the same detail.

Figure 3.4 shows a snippet of code that carries out XOR encryption on the input plaintext. As can be seen, the CFG that represents it not only shows jumps within the program but also enables loop detection. This is discussed below in Section 3.5.3, but simply a directed edge back to a already traversed basic block represents a loop.

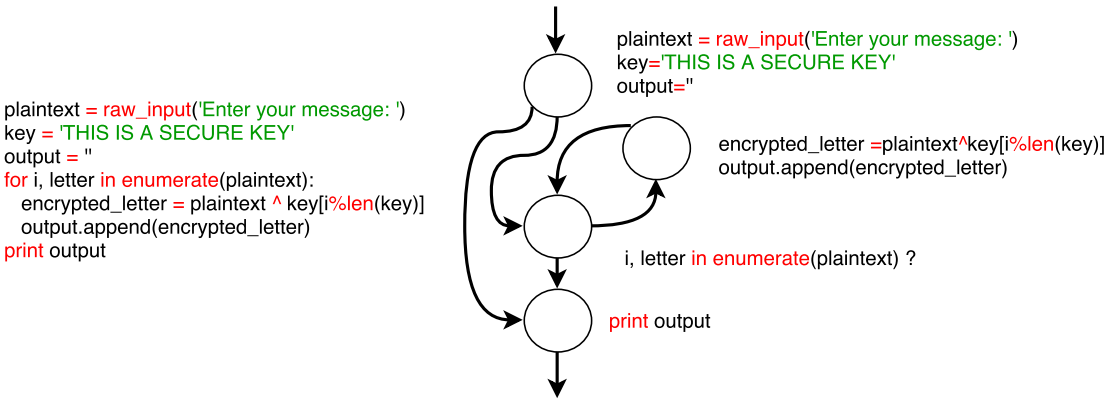

FIGURE 3.4: Relationship between Code and its CFG

Furthermore, we use the CFG to aid with finding cryptographic keys across multiple execution traces: this application is discussed in Section 3.6.2.

3.5.3 Loop Detection

A loop in an application's code is defined as a sequence of instructions that repeats until a condition has been reached. The condition usually marks the completion of a process, such as checking whether a counter has reached a given number or getting an item of data and changing it.

```
for i = r down to 1 do:
    B = ((B - S[2 * i + 1]) >>> A) ^ A
    A = ((A - S[2 * i]) >>> B) ^ B
B = B - S[1]
A = A - S[0]

return A, B
```

LISTING 3.6: RC5 Decryption

In cryptographic operations, loops are often used for the latter case. A loop will take in a piece of data, in our case encrypted data, and it will be decrypted during the loops execution. Listing 3.6 demonstrates the importance of loops to recovering data from cryptographic functions. It shows the main body of how input is decrypted with RC5, a symmetric-key block cipher. The decrypting of the data mainly happens in the `for-loop` body, and then a final linear transformation is applied afterwards.

Choosing a Loop Detection Algorithm

Loop detection is an active area of research problem and as a result there are many algorithms we can use for the task. We considered two known algorithms for this thesis, namely Tarjan's algorithm for finding dominators in a flow-graph [27] and Tubella's method for dynamic loop detection [30].

Tarjan's method has the advantage of being faster and the ability to locate dominant relationships with a flow graph. As speed is a motivation for this thesis, we prefer choosing algorithms that are faster. Furthermore, knowledge of dominant relationships is useful as it allows us to determine loop hierarchy which in turn lets us find nested loops and loop relationships.

However, Tubella's method, although slower, also allows us to infer loop hierarchy. Furthermore, we can learn how many iterations a loop body goes through, and how many times the loop is called. For example, this information is useful as if we know we have a cipher that operates on 512 bit blocks, then we can assume loops with 512 iterations may be of interest.

Tubella's Loop Detection Algorithm

Tubella's loop detection algorithm works by finding repeated executions of the same code address. If a piece of code follows a variety of different addresses, and then hits the same one twice, we can safely identify it as a loop iteration. The algorithm identifies a full loop execution when there is no jump back to that address, i.e execution continues to the next instruction, when the code jumps to a location outside of the loop or if a return statement is hit within the loop body. Figure 3.5 shows an example of this; the loop body, *B, C, D and E*, is first detected by the backwards jump from E to B. The end of the loop execution is detected after the second iteration with the jump to F.

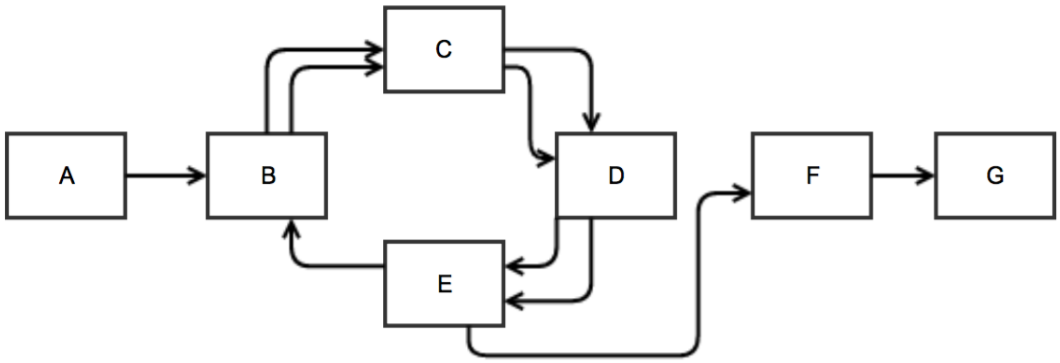

FIGURE 3.5: Detecting Loops

To track nested loops and loop statistics, a *current loop stack* is used. This allows us to track all the loops currently running, with the top of the stack being the current innermost loop. For each instruction the algorithm checks if its a jump or call and, if that address is not on the loop stack, it is pushed on so we can record the new loop execution. If the address is on the stack, we check to see if the jump or call instruction is actually taken, and if so a new iteration for that loop is recorded.

3.6 Detecting Cryptographic Loops and Recovering Data

Once loops have been detected, the next step is to differentiate between cryptographic loops and normal application loops. Applications in general contain a lot of loops, e.g. a loop in the main function body which continuously takes in and processes input from a user, and if our tool accepted all of them as cryptographic then a lot of false positives would be produced.

3.6.1 Cryptographic Loop Detection

In order to correctly identify the buffers containing cryptographic data we must be able to define characteristics of a cryptographic loop. Once we understand those characteristics, we can select the loops that fit the given criteria from the loops we detected in the program being analysed. This section covers methods we use and build upon to detect cryptographic operations.

Our approach is split into two halves: the first looks at the proportion of bitwise arithmetic operations within a loop as cryptographic loops are known to contain a greater than average ratio of bitwise arithmetic. The second then looks at how we can filter loops based on entropy. Encryption tends to increase the information entropy of loop inputs by a higher than average amount each iteration than no cryptographic loop bodies. By detecting this increase we are again able to select candidate loops.

Bitwise and Arithmetic Operations

Instruction profiling provides us with information about the run of a program which helps build up a picture of what the program does on a higher level. We know that a proportionally higher number of bitwise and arithmetic instructions are indicative of cryptographic instructions, and therefore we use this as a method of recognising candidate cryptographic loops.

To detect high arithmetic loops we go through each iteration of a loops execution and have a count for the total number of instructions, and a count for the number of bitwise and arithmetic instructions seen. Then we select loops with over 35% bitwise and arithmetic instructions. This value was chosen empirically after trial and error with different thresholds as it was found to always include the cryptographic loops with the minimum amount of extra noise.

In order to determine if a loop contains a bitwise or arithmetic operations we select those with the following types: `logical`, `shftrot`, `bit`, `binary and arith`. These types are select types from those provided for each instruction in the X86 Instruction Reference [17]. Table 3.2 consolidate this with an example showing instructions along with a classification of whether they are bitwise arithmetic or not.

Furthermore, to increase the speed of our analysis tool, we only analyse the first execution of the loop and not subsequent loop executions. This is as each loop execution will

TABLE 3.2: Instructions Identified as Bitwise Arithmetic Pperations

Bitwise Instruction	Classification	Instruction Type
sub rsp, 0x68	Arithmetic	arith
push rbp	None	NA
mov rbp, rsp	Ignored	NA
shl rdx, 0x20	Arithmetic	shftrot
mov rbx, rdi	Ignored	NA
or rdx, rax	Bitwise	logical
sub r13, qword ptr [rip+0x21e2]	Arithmetic	arith
push r15	None	NA

execute the same instructions and therefore will have the same count of arithmetic and bitwise instructions as the first execution. If the first execution is identified as potentially cryptographic by our tool, then we treat all executions as potential cryptographic.

Entropy

Entropy measures the *unpredictability of information content*. A trivial example is the outcome of a Taekwondo match, for when two equally matched fighters fight the entropy of the outcome of the match is at its highest as both are equally likely to win. If on the other hand, the fighters were unequal, and one had a greater chance of winning the outcome would have lower entropy as there would not be that much unpredictability in the outcome.

The English language in general has low entropy: this is as the subsequent letters in a word and words in a sentence are to a large extent predictable. In an English sentence, if one came across the word `in`, it is highly probable that the next word will be either `the` or `a` meaning there is low unpredictability. Similarly in a word the combination of letters `th` is more likely than `qu`, and a word is more likely to contain an `e` or an `a` rather than an `x` or a `z` [19].

We use this fact to detect cryptographic loops decrypting ciphertext into plain English. The ciphertext before decryption will have a higher entropy and contain more random data than the plaintext after decryption. Detecting this drop in entropy is what we use to select potential cryptographic loops.

Entropy is defined as shown in equation 3.1, where `P` is the probability mass function.

$$H(X) = -\sum_{i=1}^{n} P(x_i) log P(x_i) \tag{3.1}$$

Our work uses a form of *b-ary entropy*. *b*-ary entropy is defined to be of a source $\mathbb{S}=(S,P)$ where $\mathbb{S}$ is a source alphabet $\mathbb{S}=a_1 \ \ldots \ a_n$ with discrete probability distribution P over the source [11]. The source alphabet we use is of size 512, representative of the 512 byte values possible in our alphabet. Equation 3.2 is used to calculate entropy in our work.

$$H_b(\S) = -\sum_{i=1}^{512} \frac{|b|_i}{n} log_b \frac{|b|_i}{n} \tag{3.2}$$

A final processing step is needed after calculative entropy because the length of our inputs and outputs may be different, and we need to compare entropy values. As a result after calculating entropy, we follow Lutzs approach and scale our entropy values to fall between 0 and 1 [18] and select loops with over a 20% entry decrease.

3.6.2 Recovering Cryptographic Data

Once potential loops have been narrowed down, memory reconstruction techniques are applied to potential inputs, cryptographic keys and outputs. This allows for the data to be recovered from the execution trace and to be presented in a human readable format. Then our last filter can be applied which allows our tool to correctly identify the key and the plaintext. Our novel approach to recovering the encryption key by statistically correlating multiple execution traces for the same application increases the confidence we have in our results.

Memory Reconstruction

Once we have identified potential cryptographic inputs, keys and outputs we need to verify them. The traces we generate, as discussed in 3.4, already contain information about memory reads and writes. For memory reads we store the memory address read from and the hexadecimal value it read, and similarly for writes we store the memory address written to and the hexadecimal value it wrote.

The first task is to convert the hexadecimal output back into printable ASCII that we can understand. This will mean any plaintexts we are trying to read will be understandable, and if we have a known plaintext to compare to we can verify that we are looking at the right memory locations. Secondly, as a program only has a set amount of memory assigned to it, it is common to see multiple writes to the same memory location. An example of when this can occur is when a variable is overwritten in the program and its value is updated. The variable is stored at the same location in memory: therefore, the value stored there is overwritten with the new value and the old value is lost.

```
1 0x7ffe7e21c4a5: ['r 0x41', 'w 0x23', 'r 0x23', 'r 0x23']
2 0x7ffe7e21c4a6: ['r 0x41', 'w 0x22', 'r 0x22', 'r 0x22']
```

LISTING 3.7: Sample Memory Accesses during an XOR Application Run

Listing 3.7 shows an example of read and writes seen in a XOR application used for evaluation generated by our tool to analyse the access patterns to a specific memory address. The value to the left of the `:` is the memory address, and the list afterwards show the reads, *r*, and writes, *w*, in the order they were made along with the value.

The encryption input to this example is the string `AAAAAAAAAAAAA`. The letter `A` in hexadecimal is represented by the value `0x41`. Lines 1 and 2 in Listing 3.7 show that this

value is read from the respective addresses and then subsequently overwritten with `0x22` and `0x23` respectively. These two values correspond to `B` and `C`, which are the letters those characters were encrypted to by the application.

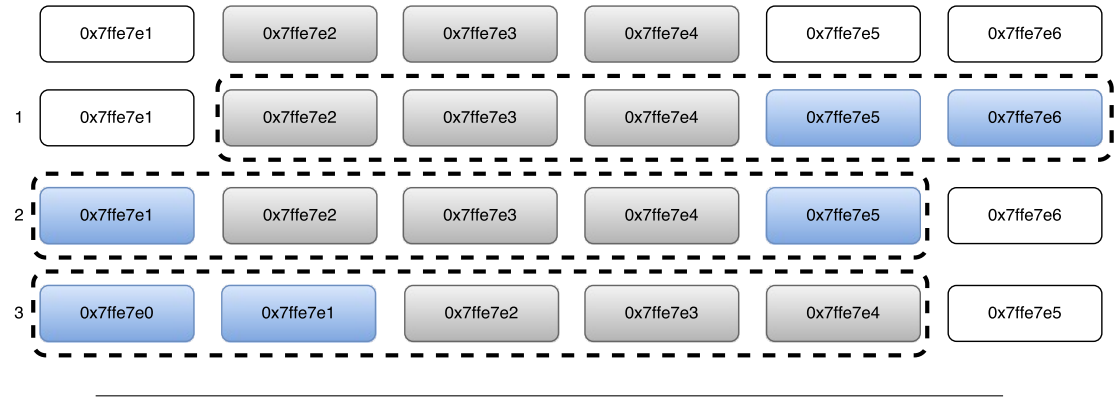

FIGURE 3.6: Memory Address Sliding Windows

The memory addresses are consecutive and when searching for keys which maybe of fixed length sizes such as 512 bits we can search for sliding windows of that size around values that are known to be in the key. Figure 3.6 demonstrates this idea. Here we are searching for a key of length 5 and we know that the three consecutive memory addresses `0x7ffe7e2, 0x7ffe7e3` and `0x7ffe7e4` are part of the key. We can then assume that the key will be one of the numbered cases depicted. This works as we assume in this thesis that cryptographic data will be stored at consecutive memory addresses.

Statistical Trace Analysis

The previously discussed entropy measures provide us with sound tools to recover the cryptographic inputs and outputs. However cryptographic functions often have more than one relevant input, the key as well as the plaintext to be encrypted. This section discusses our motivation for analysing multiple traces and Chapter 4.4.3 discusses the approach taken.

```
$ pypy majorkey trace.out
** logging output omitted **
potential inputs
_____________________________________________
0x41 0x62 0x63 0x64 0x61 0x62 0x63 0x64 0x61 0x62
0x63 0x69 0x61 0x62 0x63 0x64 0x61 0x62 0x63 0x64
Abcdabcdabciabcdabcd
_____________________________________________
_____________________________________________
0x54 0x68 0x65 0x79 0x20 0x64 0x6f 0x6e 0x27 0x74
0x20 0x77 0x61 0x6e 0x74 0x20 0x75 0x73 0x20 0x74
0x6f 0x20 0x67 0x65 0x74 0x20 0x61 0x20 0x66 0x69
0x72 0x73 0x74 0x2e 0x2e 0x2e
They don't want us to get a first...
_____________________________________________
potential outputs
0x15 0xa 0x6 0x1d 0x41 0x6 0xc 0xa 0x46 0x16 0x43
0x1e 0 0xc 0x17 0x44 0x14 0x11 0x43 0x10 0x2e
0x42 0x4 0x1 0x15 0x42 0x2 0x44 0x7 0xb 0x11
```

```
0x1a 0x15 0x4c 0x4d 0x4a
```

LISTING 3.8: MajorKey XOR Output Trace 1

To illustrate this point Listing 3.8 shows the output produced by MajorKey for an XOR application. The potential inputs have been correctly identified as `Abcdabcdabciabcdabcd` and `They don't want us to get a first...`, however we are unable to determine which is the key and which is the plaintext.

```
vagrant@Derp3: pypy majorkey trace.out
** logging output omitted **
potential inputs
__________________________________________________
0x41 0x62 0x63 0x64 0x61 0x62 0x63 0x64 0x61 0x62
0x63 0x69 0x61 0x62 0x63 0x64 0x61 0x62 0x63 0x64
Abcdabcdabciabcdabcd
__________________________________________________
0x53 0x6f 0x20 0x77 0x68 0x61 0x74 0x20 0x77 0x65
0x20 0x67 0x6f 0x6e 0x20 0x64 0x6f 0x2c 0x20 0x67
0x65 0x74 0x20 0x61 0x20 0x66 0x69 0x72 0x73 0x74
0x20 0x3b 0x29
So what we gon do, get a first ;)
__________________________________________________
potential outputs
0x12 0xd 0x43 0x13 0x9 0x3 0x17 0x44 0x16 0x7 0x43
0xe 0xe 0xc 0x43 0 0xe 0x4e 0x43 0x3 0x24 0x16 0x43
0x5 0x41 0x4 0xa 0x16 0x12 0x16 0x43 0x52 0x48
```

LISTING 3.9: MajorKey XOR Output Trace 2

Similarly from looking at listing 3.9 we are able to determine that the two potential inputs are `Abcdabcdabciabcdabcd` and `So what we gon do, get a first ;)`. Looking at just this listing in isolation still provides no clue as to which of the two inputs is the plaintext, and which the key; however by analysing commonalities between the traces we can learn which is which.

In the case above, we learn that the key is `Abcdabcdabciabcdabcd` and the plaintexts are `They don't want us to get a first...` and `So what we gon do, get a first ;)` for listings 3.8 and 3.9 respectively.

This approach works trivially for an easy encryption algorithm such as `XOR` but for more complicated forms of encryption such as `DES` it is a little less clearer. `DES` uses substitution boxes, S-Boxes, to conceal relationships between the key and the ciphertext. As such we have to ensure that we do not mistake cryptographic constants, such as S Boxes, as the key. We do this by whitelisting against known constants that appear in common implementations. Furthermore, we can rule out all inputs that are of incorrect key size for the given encryption method, if it is known.

Chapter 4

Implementation

4.1 Overview

This chapter will cover relevant implementation details of MajorKey. MajorKey is split into three main tools:

- Instrmentation tool.
- Analysis tool.
- Keyfinder tool.

We discuss MajorKey's Pintool in Section 4.2 and how it is built on top of the Pin framework. Then an overview of MajorKey's analysis tool is provided in Section 4.3. Section 4.4 explains the loop selection and filtration methods, and how we made them interactive to users. It also explains our Keyfinder tool which is how we compare multiple traces to narrow down the encrpytion key. Finally, Section 4.5 highlights some of the interesting optimisations we made to increase the accuracy and speed of our tool.

4.2 MajorKey's PinTool

The implementation of our Pintool is based on processing `Instructions` and `Memory` at a instruction level granularity. The next sections explain further how we do this.

There are two ways to modify a program with a Pintool: through *instrumentation* or *analysis*. Adding an instrumentation routine to a Pintool will define where instrumentation is inserted to an application. It occurs once per object and does the *heavy lifting* of the Pintool. Analysis routines define what the Pintool should do once instrumentation has started and are run every time an object is accessed. MajorKey's Pintool uses both, instrumentation routines to define what to instrument, and analysis routines defining what happens once instrumentation occurs.

Instrumentation

Pin instruments in a *just-in-time* manner. This means a copy of the application is created on the fly and run the original application code is never executed. We take advantage of this and our Pintool can skip sections of the application run and start instrumenting from a set point speeding up the time taken to run as less code is being instrumented. Similarly our tool can start instrumentation from a given memory address or function name that can be passed in as a command line argument.

Analysis

Instrumentation is performed at an instruction level granularity, and for each instruction we call two analysis routines, `InstructionTrace` and `MemoryTrace`. `InstructionTrace` is run on every instruction, and the memory address of the instruction, routine and image information, and disassembly information is written to file as specified in Chapter 3.4. `MemoryTrace` is called on every instruction that reads or writes to memory. Again as described in Chapter 3.4 it logs information about the value read/written and the memory location accessed.

4.3 MajorKey Python Framework

Our tool is written in Python and split up into four main modules as shown in Figure 4.1. This section covers our language and design choices.

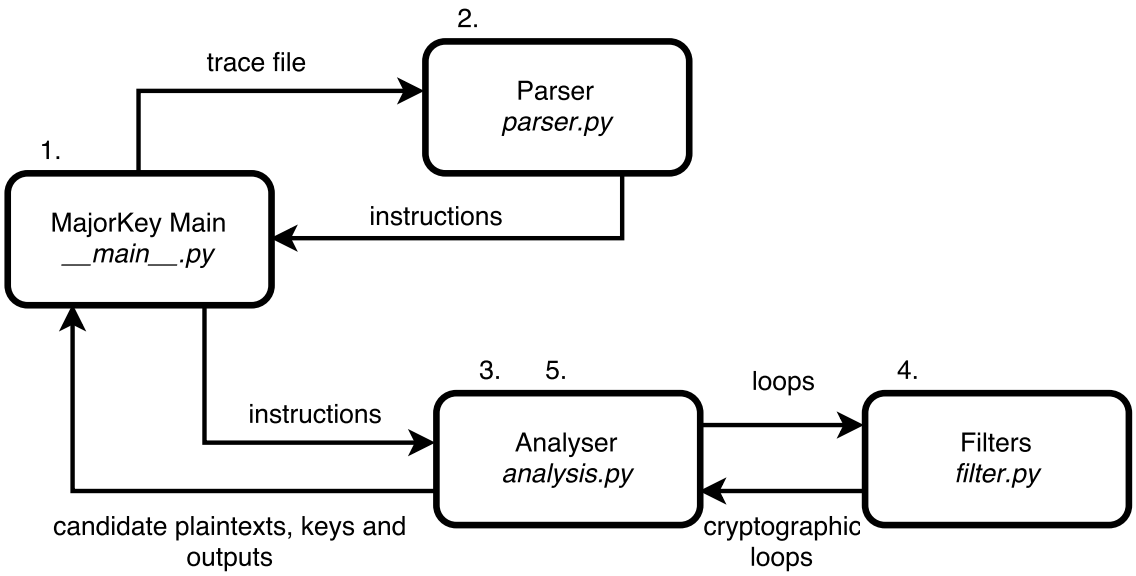

FIGURE 4.1: MajorKey Pipeline

The MajorKey pipeline is shown in Figure 4.1. Everything starts at `1` and the initial input, the trace file, is passed to `2`. The Parser parses the trace file into a format that the rest of

the tool understands, and passes back a list of `Instruction` objects to the Main. At `3` the instructions are passed to the Analyser which detects basic blocks, the control flow of the program and loops. The loops are then passed into `4` where the filters are run over the loops independently. Potential cryptographic loops are detected and passed back to the Analyser at `5`. The Analyser then combines the results and generates weightings for potential plaintexts, keys and outputs for loops identified as cryptographic. The potential plaintexts and keys are converted back into ASCII, a human readable form, and displayed to the user.

We chose to split up the tool in such a form to make it easy to use and extend. It is simple to add new filters to the Filter module, and similarly easy to switch out any of the above modules to change or add to the functionality of the tool. An example of where this may be useful is replacing the current Python Analysis module with a C module to for speed ups.

4.3.1 Python: Language of Choice

Python is an easy to use language that provided us with the ability to start developing fast, and has simple, explicit syntax. It is easy to run on Linux, Windows and OS X therefore allows us to develop a cross-platform tool. Furthermore the language has a strong community built around it with lots of public libraries. `argparse` is an example of one and it allowed us to easily handle command line options as covered in Chapter 4.4.1.

4.4 Loop filtering

Each filter method used to whittle down the cryptographic loops from the application loop set is applied independently, and in parallel, and then the results are combined in a map reduce fashion. The filters are applied separately to avoid losing potential inputs or outputs due to one filter not recognising the data as important but others do. Afterwards, a higher weighting is applied to inputs and outputs that appear in the output of more than one filter.

4.4.1 User Controllable Knobs

MajorKey is an automated tool that can take in a trace and output potential inputs and outputs without requiring any user assistance. However it aims to be flexible and as such for each filter we provide the ability to adjust it when the program is run.

Filters can be adjusted at run-time by passing in alternative values for the filter thresholds. Each filter has given defaults for the filters that have been empirically chosen but there is an option to override them. Listing 4.1 shows the thresholds that a user can adjust at run-time, along with an explanation of what they do. Having this ability means a user can adjust the given thresholds without the need to modify the source code of our tool.

```
usage: majorkey [-h] [--debug DEBUG] [--active_threshold ACTIVE_THRESHOLD]
                [--iteration_threshold ITERATION_THRESHOLD]
                [--arithmetic_threshold ARITHMETIC_THRESHOLD]
                [--entropy_threshold ENTROPY_THRESHOLD]
                filename

MajorKey :raised_hands:

positional arguments:
  filename              Trace file to be loaded in

optional arguments:
  -h, --help            show this help message and exit
  --debug DEBUG         If true, will open an interactive IPython
                        session after loop detection
  --active_threshold ACTIVE_THRESHOLD
                        Set's active_threshold for active_loops.
                        Default value is 20
  --iteration_threshold ITERATION_THRESHOLD
                        Set's iteration_threshold for
                        high_arithmetic_loops. Default value is 20
  --arithmetic_threshold ARITHMETIC_THRESHOLD
                        Set's arithmetic_threshold for
                        high_arithmetic_loops. Default value is 0.1
  --entropy_threshold ENTROPY_THRESHOLD
                        Set's entropy_threshold for
                        high_entropy_loops. Default value is 4

Bless up
```

LISTING 4.1: MajorKey Usage

We evaluated our tool against a variety of applications, including an obfuscated XOR program [1]. The knobs proved useful during the evaluation of the obfuscated XOR program, by lowering the threshold from 20 to 8 for the `iteration_threshold` filter meant the encryption loop was not skipped. Had we not been able to lower the threshold, the loop would have evaded detection as the obfuscations applied to the program resulted in a decreased number of reduced loop iterations.

4.4.2 Live Debugging Mode

Another feature we chose to implement is a live debugging mode. It provides a simple interface to view and manipulate the loops detected by our work. It can be activated by setting the command line option, `-debug=True`.

Listing 4.2 shows the interactive Python session that gets started if the program is called with `-debug=True`.

```
vagrant@Derp3:~$ pypy majorkey ../trace.out --debug=True
*** output omitted ***
time taken: 25.1617631912

Python 2.7.3 (2.2.1+dfsg-1ubuntu0.3, Sep 30 2015, 15:18:40)
Type "copyright", "credits" or "license" for more information.

```

[1] Our tool is not specifically designed to work with obfuscated programs

```
8  IPython 4.2.0 -- An enhanced Interactive Python.
9  ?         -> Introduction and overview of IPython's features.
10 %quickref -> Quick reference.
11 help      -> Python's own help system.
12 object?   -> Details about 'object', use 'object??' for extra details.
13
14 In [1]: outs[0].executions
15 Out [1]: <{[['0x00007f51832e3c6b': [[<<class 'instructions.BB'> at
16                          0x7ff44489c608>,
17    <<class 'instructions.BB'> at 0x7ff44493c8a8>],
18   [<<class 'instructions.BB'> at 0x7ff444923440>,
19    <<class 'instructions.BB'> at 0x7ff444922c98>,
20    <<class 'instructions.BB'> at 0x7ff444932410>,
21    <<class 'instructions.BB'> at 0x7ff444933da8>,
22    <<class 'instructions.BB'> at 0x7ff4449479b8>]
23    ...
```

LISTING 4.2: MajorKey Interactive Debug Mode

The loops detected are stored in the variable `outs`, and this provides access to information about loop executions, as shown on line 15 where the beginning of the representation of the loop executions is printed. Once a debugging session has been started further modules such as the `filters` module can be loaded into to run functions within them or new functions can be written and tested over live input data. We have chosen to provide this feature as it enables a user to interact with the data we process instead of just receiving output from the tool. It provides a clean easy to user interface without the need for the user to understand the source code or touch any files.

4.4.3 MajorKey's Keyfinder Tool

As discussed in Section 3.6.2 it can be hard to differentiate between the key and the input message so we analyse multiple traces to increase our confidence. If the same key has been used then it will be constant across runs, so we can assure that its value will stay constant and therefore be distinguishable from the varying plaintexts. Note: The key is not usually stored within a program therefore will not be the same for every message sent and recevied, however it is common for the key to be the same across a session in which multiple messages can be sent and received.

Figure 4.2 shows how input traces are passed through our tool and then combined. First, each trace is analysed individually to generate a list of potential input texts and keys. This is represented by the `trace_*.out` files being passed into the MajorKey tool and producing a respective `trace_*.json` file.

The output to the `JSON` files by our tool is the same as the results printed out to the user after MajorKey has run. However, the results are not dumped straight to a file but instead have been parsed into `JSON`.

```
1 a)
2 vagrant@Derp3:$ pypy majorkey ../xor.out -
3                 -keyfinder='keyfinder_dumps/xor.out'
4 * output omitted *
5 Confidence: 0.88
6 ------------------------------------------------
7 potential inputs
```

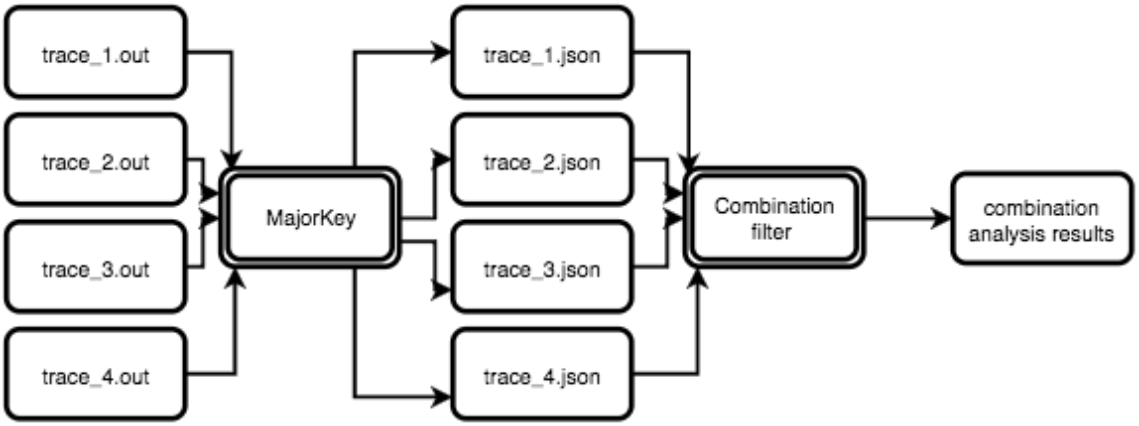

FIGURE 4.2: Keyfinder Pipeline

```
0x41 0x62 0x63 0x64 0x61 0x62 0x63 0x64 0x61 0x62 0x63 0x69
0x61 0x62 0x63 0x64 0x61 0x62 0x63 0x64
Abcdabcdabciabcdabcd
0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
AAAAAAAAAAAAAAAAAAAAAAAA
potential outputs
0 0x23 0x22 0x25 0x20 0x23 0x22 0x25 0x20 0x23 0x22 0x28 0x20
0x23 0x22 0x25 0x20 0x23 0x22 0x25 0 0x23 0x22 0x25 0x20
------------------------------------------------

b)
[
    {
        "confidence": 0.88,
        "data": [ "0x41",  "0x62", "0x63", "0x64", "0x61", "0x62",
                  "0x63", "0x64", "0x61", "0x62", "0x63", "0x69",
                  "0x61", "0x62", "0x63", "0x64", "0x61", "0x62",
                  "0x63", "0x64"]
    },
    {
        "confidence": 0.88,
        "data": ["0x41", "0x41","0x41", "0x41","0x41", "0x41","0x41",
                 "0x41", "0x41","0x41", "0x41","0x41","0x41","0x41",
                 ...]
   },
   ...,
]
```

LISTING 4.3: MajorKey Keyfinder Output

Listing 5.9 shows an example of the JSON that would be generated for the output depicted at `a)`. We use `JSON` because it supports key-value pairs and has fast, easy to use Python modules readily available.

Once all the traces that are being compared have been joined together, we pass them to our final filter, `Keyfinder`, which reads in all the `JSON` files in the directory passed to it and looks for overlap between potential inputs. We applied the longest consecutive subsequence problem to each pair of inputs to find a match for the key, but if less than eight characters matched we stop looking as it is less likely to be a key match and more

an overlap in plaintext messages. Weightings between 0 and 1 are then applied, with a 1 being a complete match.

After this analysis, we re-calculate the confidence for the inputs being the key using the original confidence passed in, and the weightings generated during analysis. The results are then sorted, and the top choices displayed accordingly.

We chose to take this approach as it provided flexibility. All the traces for an application can be passed into `keyfinder` or a specific combinations to detect further patterns. However a downside is when applying the longest consecutive substring problem to the texts if the plaintexts have long matching substrings, we are more likely to incorrectly identify them as the key. Currently we do not avoid this, but we could by blacklisting certain strings and ignoring them if they appear.

4.5 Optimization

We chose to make speed one of our goals as current tools that exist are fairly slow. In Grobert's evaluation he noted for example that runtimes for analysing `RC4` were in the order of 15 minutes [12]. Although malware analysis can be done offline and time isn't a huge constraint, we recognised that large speed ups were possible. Some factors we considered to reduce our time are documented below.

4.5.1 Binary Instrumentation

Binary instrumentation can be timely, especially when the tool is applying memory tainting techniques. Initially, MajorKey worked with traces generated by an existing Pintool framework that provided dynamic binary instrumentation [24]. However, it was too slow and some traces were taking over an hour to generate. For example, a GPG encrypt operation which takes less than a minute when running without instrumentation was taking nearly an hour with instrumentation. If instrumentation takes excessively long with respect to the original runtime of the program, there is a chance that the program may react differently. Balzarotti, for example, noted that there have been cases where sophisticated malware can detect its environment and if it's aware it is being instrumented or run in an emulator it can change its execution path on the fly [2].

As such we chose to write our own Pintool as discussed in Chapter 4.2. The advantage of writing our own tool instead of using an existing heavy duty one out there was the ability to keep instrumentation times on par with the original application runtimes. Furthermore, our tool is specific to its function therefore has a relatively simplistic codebase which is easy to maintain and modify.

4.5.2 Parallelising Parsing

After binary instrumentation we end up with a large trace which can be over a gigabyte in size. Processing the entire file and searching through the trace for basic blocks and

TABLE 4.1: Instruction Analysis From Traces

Application	Max BB size	Max loop size
XOR	165	1342
DES	204	6611
RC4	293	2209
RC6	243	5337
Obfuscated XOR	135	1345

loops, became extremely tedious and slow. This was due to our application requiring so much memory it eventually started to use swap memory. Using swap means instead of accessing RAM memory which has access times of roughly 50 nanoseconds, the program was accessing memory on the machine's hard drive which is roughly 10000 times slower [1].

One method of avoiding this would be to read in a chunk of the file, process it, store the results, then read in the next chunk, process that and continue until we reached the end of the file. Processing data in a linear fashion like so takes a long time for the size of traces we generate, as such we recognised the opportunity to parallelise the process.

4.5.3 Splitting Strategy

The trace generated by the binary instrumentation tool contains the instructions executed by our program in a chronological order. We take advantage of the spatial and temporal relationship: this helps decide where to separate the trace into chunks so we can parallelise our tool.

Table 4.1 shows relevant information about the programs we evaluated pertaining to the size of basic blocks, BBs, and loops. As can be seen, the largest size of a BB is 293 instructions, and the largest loop size across applications tested is 6611 instructions.

When splitting our trace into multiple chunks, we have to ensure that we do not miss any of those by splitting the trace up such that a basic block or a loop ends up with half its body in one chunk and the other half in another chunk. If this happened we could end up missing a cryptographic loop itself or a basic block. As basic blocks are used to build the CFG, which in turns guides our loop detection algorithm, a mistake here could cause our results to be incorrect.

Using the information from the table we chose to parse our file in chunks of 750,000 instructions with a 100,000 size overlap. The overlap means that those instructions at a chunk boundary will appear in both chunks to either side of it, so a loop cannot be missed. Figure 4.3 shows how this works in practice with the trace on the left being split into three chunks. In chunk two in the diagram contains the end of chunk one and the beginning of chunk three.

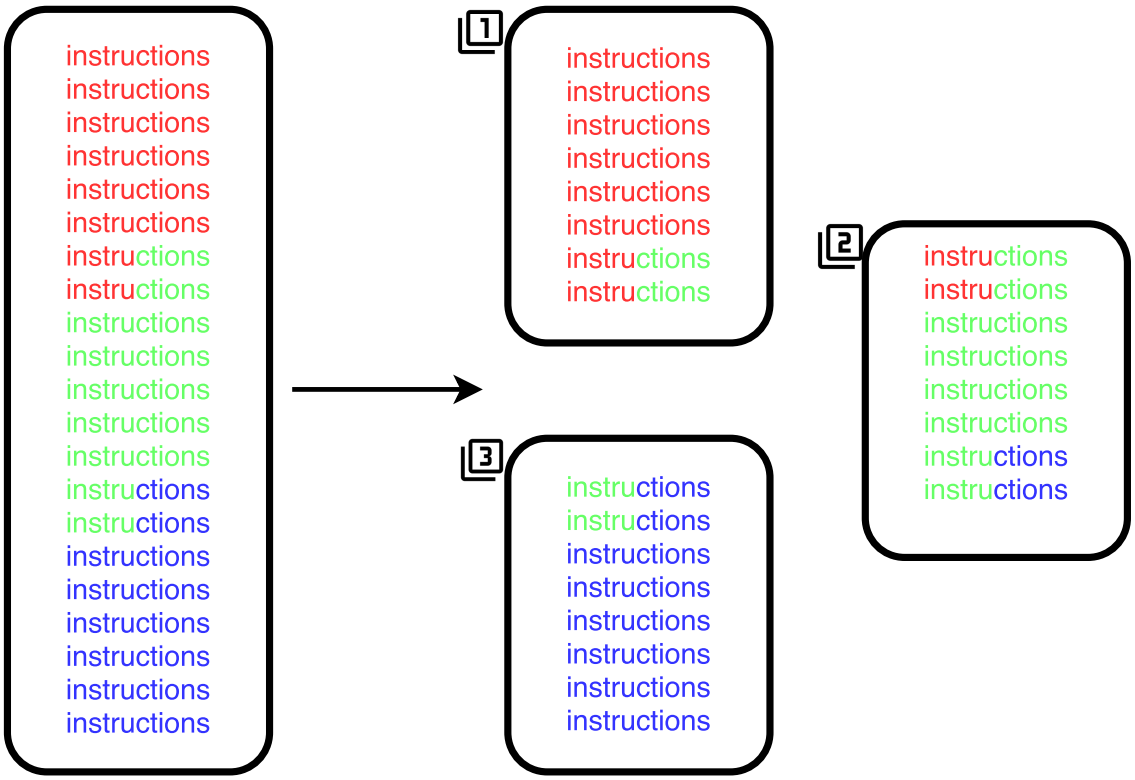

FIGURE 4.3: MajorKey : Splitting Strategy

4.5.4 Multiprocessing Module

The traces generated by the binary instrumentation tools are a chronological list of the instructions executed by the target application. As such we can take advantage of the spacial and temporal locality of instructions and process the trace file in chunks. Furthermore, if we parallelise this operation we can see large time speed ups.

There are two main Python modules used for parallelisation, threading and multiprocessing. The Threading module is designed to keep an application responsive whilst background tasks are going on, such as a handling connection to a database whilst accepting user input. The database connection task is moved into a different thread whilst the main thread is kept free so the application remains fast. Due to the Python global interpreter lock, GIL, these two operations don't actually happen in parallel but CPU time is split between the two threads.

The other is the Multiprocessing module, which is for when executing more than one action at a time is actually required. Instead of placing tasks in their own thread that has access to the CPU when another thread is idle, a Multiprocessing process can run on its own CPU and therefore more efficiently. A disadvantage of spawning extra processing however is the I/O overhead it introduces as data is being moved between the different processors consequentially which may increase the total runtime.

MajorKey uses the Multiprocessing module as *real* parallelism is actually required for loop detection and filtering of the different trace chunks. As no data is shared between processes, our application receives significant speedups for applications that generate

TABLE 4.2: PyPy Speed Ups

Application	Python runtime	PyPy runtime
XOR	2m 59s	49s
DES	24m 48s	7m 10s
RC4	4m 33s	1m 32s
RC6	6m 02s	1m 50s
Obfuscated XOR	2m 12s	30s

large trace files: this is discussed further in Section 5.3.1 in our evaluation.

4.5.5 PyPy Usage

We chose to write MajorKey in Python due to its ease of use as discussed in Section 4.3.1. However, running our tool with Python was not the fastest option as such we looked at PyPy as an alternative [4]. Note: MajorKey is not limited to PyPy and can be run with Python, Cython or any other variant, but choose to run our tool with PyPy due to the speed-ups it provides.

PyPy works the same as Python does outwardly, but has a different internal implementation. These features provide various advantages that made PyPy appealing to this work including:

- Speed: PyPy uses a Just-in-Time compiler, and therefore Python programs often run faster on PyPy.
- Memory Usage: PyPy has better memory management and therefore uses several hundred MBs less memory than would be required if we ran our tool with Python.
- Compatibility: Most popular Python libraries work with PyPy and therefore we are able to use standard libraries without having to worry about any potential breakages.

As our tool relies entirely on Python code, as opposed on underlying C code, PyPy is able to optimize it and provide sizable speedups. Table 4.2 shows the running time of our tool against different applications in Python and in PyPy. For each application we tested, PyPy provided over a 50% time speedup.

Chapter 5

Evaluation

MajorKey is a cross-platform suite of tools, as it runs on Pin and Python it will work on Linux, Windows and OSX. It successfully works will all 3 major operating systems, however for the purpose of evaluation we focus on Linux only. The evaluation environment is outlined in Section 5.1.

In Chapter 1.2 we state our tool makes the following contributions:

1. An architecture to recover cryptographic material from executable binaries.
2. A Pintool to generate traces for target binaries.
3. A Python framework, and tool, that analyses said traces to recover cryptographic material.
4. A new technique that aids in narrowing down potentially relevant cryptographic material.
5. A parallelised approach speed up in trace parse and analysis time.
6. The recovery of cryptographic material from applications carrying out XOR, DES, RC4 and RC6 operations.
7. Recovery of cryptographic material for the RC6 encryption algorithm, which has not been done before.

We begin by assessing *claim 1 and 7*, and provide an example of how our tool works in Section 5.2. This allows the reader to understand how results are delivered by the tool and shows we have successfully created a tool capable of achieving the remaining goals of this thesis.

In order to evaluate our claim that we can recover cryptographic material, *claim 2, 3, 4 and 6*, Section 5.3 shows the results of testing MajorKey against multiple applications to recover the keys, plaintexts and outputs. This section also provides analysis results for `Obfuscated XOR`, even though we did not directly target our tool to work with obfuscated programs, an advantage of our method is it's ability to work with simple obfuscation. Furthermore, we evaluate the confidence scores of MajorKey's results alongside the ratio of false positives.

Section 5.3.1 then looks at *claim 5* and analyses the times taken for our instrumentation tool to run the application, and compares the added instrumentation time to the original running time of the application. We are able to keep our instrumented running time within 5% of the initial time taken by the application.

MajorKey is then evaluated against existing tools in Section 5.4. Here we provide a quantitative comparison between the results our tool achieved and results achieved by other tools.

Finally, in Section 5.5, we present a case study of MajorKey with an chat application that carries sends encrypted messages using `RC6`. This further evaluates *claim 7* and shows that MajorKey is capable of working with complex applications.

5.1 Evaluation Environment

For this evaluation the tracing and the MajorKey tool have both been run on a 64 bit Ubuntu VirtualBox, more precisely `Ubuntu 14.04.3 LTS (GNU/Linux 3.13.079-generic x86_64)`. This virtual machine is being run on a laptop running Mac OS X 10.11.5. The virtual machine has 1024 MB of RAM and runs on one core of the host machine. The version of Pin being used is Pin 2.14-71313 with GCC version 4.4.7. To run our tool we use Python 2.7.3 and PyPy 2.2.1 with GCC 4.8.4. This environment remains consistent for the entirety of the evaluation.

5.2 Walk Through Example: RC6

We begin by walking through the analysis of a RC6 decryption. RC6 was an AES finalist. RC6 has not been tested before for recovery of cryptographic data therefore this thesis presents it as a novel result. We test our tool with three different implementations of RC6, in the walk through example we use the `tinycrypt` implementation with 20 rounds of encryption and a 256 bit key [20]. Only one 128 bit block of input is encrypted to make analysis easier to follow.

```
1 char *test_keys[] =
2 { "0123456789abcdef01122334455667788 99aabbccddeeff01032547698badcfe" };
3
4 char *test_plaintexts[] =
5 { "02132435465768798a9bacbdcedfe0f1" };
6
7 char *test_ciphertexts[] =
8 { "c8241816f0d7e48920ad16a1674e5d48"};
```

LISTING 5.1: RC6 Input Parameters

```
1 ./../../../pin.sh -t obj-intel64/MyPinTool.so -- ~/examples/tinycrypt/block/rc6
    /./a.out
```

LISTING 5.2: RC6 Instrumentation Command

Our RC6 application is run with the key and plaintext shown in Listing 5.1. The test key is 256 bits, and the plaintext and expected ciphertext are 128 bits. First the application

is run through our instrumentation tool using the command shown in Listing 5.2. The first parameter is the path to our Pintool, then the path to our Pintool followed by the program we want to execute, in our case the program carrying out the RC6 encryption. The trace results are saved to a file `trace.out` in the directory the command was called from.

```
pypy majorkey trace.out --keyfinder='majorkey/keyfinder_dumps'
```

LISTING 5.3: How To Call MajorKey With the RC6 Trace

The trace generated is then passed to MajorKey using the command shown in Listing 5.3, where we pass in the parameter `keyfinder` with a location so we can store the results and later run the `keyfinder` tool. The first analysis step is to parse the trace into `Instruction` objects: for our example, 160152 `Instruction` objects were created. From these instructions, the basic blocks are detected and a control flow graph is generated to show the flow of the application: for our application, there were 30062 basic blocks. **AppendixB** shows the control flow graph generated for our example application. It contains 1944 vertices that represent the unique basic blocks detected in the previous stage, and the edges show which basic blocks call each other.

```
1  Confidence: 1.0
2  ------------------------------------------------
3  potential inputs
4  0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x61 0x62 0x63 0x64 0x65 0x66 0x30
       0x31 0x31 0x32 0x32 0x33 0x33 0x34 0x34 0x35 0x35 0x36 0x36 0x37 0x37 0x38
       0x38 0x39 0x39 0x61 0x61 0x62 0x62 0x63 0x63 0x64 0x64 0x65 0x65 0x66 0x66
       0x30 0x31 0x30 0x33 0x32 0x35 0x34 0x37 0x36 0x39 0x38 0x62 0x61 0x64 0x63
       0x66 0x65
5  123456789abcdef011223344556677889aabbccddeeff01032547698badcfe
6  0x30 0x32 0x31 0x33 0x32 0x34 0x33 0x35 0x34 0x36 0x35 0x37 0x36 0x38 0x37 0x39
       0x38 0x61 0x39 0x62 0x61 0x63 0x62 0x64 0x63 0x65 0x64 0x66 0x65 0x30 0x66
       0x31
7  02132435465768798a9bacbdcedfe0f1
8  potential outputs
9  0x63 0x38 0x32 0x34 0x31 0x38 0x31 0x36 0x66 0x30 0x64 0x37 0x65 0x34 0x38 0x39
       0x32 0x30 0x61 0x64 0x31 0x36 0x61 0x31 0x36 0x37 0x34 0x65 0x35 0x64 0x34
       0x38
10 c8241816f0d7e48920ad16a1674e5d48
11 ------------------------------------------------
```

LISTING 5.4: RC6 Cryptographic Output

Listing 5.4 shows the results detected by our tool. Comparing lines 5 and 7, and line 8 and 10 from Listings 5.1 and 5.4 respectively we can see that it has correctly identified the input, and the output. It has correctly identified the key but has missed the leading 0 in the key, this can be seen by comparing lines 2 and 5 of Listings 5.1 and 5.4. Our tool detected the key but only a partial as it missed the first character. Furthermore as shown on line 1 of Listing 5.4, a confidence score of one was assigned as all of our filters recognised the loop as cryptographic. As discussed in Section 4.4.3, confidence scores are assigned by combining the weighting given from each of our filters. If all the filters recognise that loop to contain cryptographic data, then a high confidence is given to all data in that loop.

```
1  5e84356b, a8a83848
2  6002659b, 1e729f9a
3  8dda4fbf, 1c924934
4  99765bd9, fe4244d8
5  dfd1a127, 73f747eb
```

```
37482596, 57dffb86
dabf139f, 3023742c
39612c4, 3276050e
f6b36779, e11395bd
220a8b86, ea48ed81
2322c9f9, b7dbb70a
a215b2ca, d944f68
5f934c5f, b50191d6
2da353ec, 9134331
f7e8c0c, 7321e9a0
6ff2458c, 3311b59e
dd355af7, 5b05e64a
430caa83, d526156c
afbcdbd3, a58d40b6
fef4d10b, fa90d1ed
```

LISTING 5.5: RC6 Round Keys and Intermediate Ciphertext

```
['0xa8a83848', '0x6002659b', '0x1e729f9a', '0x8dda4fbf', '0x1c924934', '0
    x99765bd9', '0xfe4244d8', '0xdfd1a127', '0x73f747eb', '0x37482596', '0
    x57dffb86', '0xdabf139f', '0x3023742c', '0x39612c4', '0x3276050e', '0
    xf6b36779', '0xe11395bd', '0x220a8b86', '0xea48ed81', '0x2322c9f9', '0
    xb7dbb70a', '0xa215b2ca', '0xd944f68', '0x5f934c5f', '0xb50191d6', '0
    x2da353ec', '0x9134331', '0xf7e8c0c', '0x7321e9a0', '0x6ff2458c', '0
    x3311b59e', '0xdd355af7', '0x5b05e64a', '0x430caa83', '0xd526156c', '0
    xafbcdbd3', '0xa58d40b6', '0xfef4d10b', '0xfa90d1ed', '0x485d4e67']
```

LISTING 5.6: RC6 Cryptographic Output Continued

As well as detecting the master key, plaintext and ciphertext, MajorKey is able to pick up intermediate round keys and ciphertexts. **AppendixC** shows the full extra output generated by our tool, with Listing 5.6 and 5.5 showing the important output and 20 round keys, and intermediate ciphertexts for our RC6 application, respectively.

So far, for our example trace we have recovered the plaintext, master key, ciphertext, intermediate round keys and intermediate ciphertexts. Yet, as shown in Listing 5.4, we are still unable to differentiate between the plaintext and the key. To achieve this we rerun the trace with a different plaintext and ciphertext. Two traces will provide us with enough information to run the `keyfinder` tool to successfully determine which of the inputs is the key and which are the plaintext. This assumes that the key being used is the same: in practice this is normally the case and therefore will work. However, when it is not, our tool fails to differentiate between the plaintext and key with a reasonable amount of certainty.

```
char *test_keys[] =
{ "0123456789abcdef011223344556677899aabbccddeeff01032547698badcfe" };

char *test_plaintexts[] =
{ "00000000000000000000000000000000" };

char *test_ciphertexts[] =
{ "0ba6350d974a4a20eff199ec09f8376d"};
```

LISTING 5.7: RC6 Input Parameters

Listing 5.7 shows the second input provided to the RC6 application being tested. The key remains constant but the plaintext and ciphertext has changed. MajorKey is then run again with same command, as shown in Listing 5.3 and the results are saved to a `JSON` file which will be used by the `keyfinder` tool.

TABLE 5.1: Overview of Testing Applications

Algorithm	Implementation	Version
XOR	Custom	1.0
Obfuscated XOR	Custom	1.0
RC4-CBC	OpenSSL	1.0.1f
DES-CBC	OpenSSL	1.0.1f
DES-CBC	Tareque Hossain	1.0
DES-CBC	Tinycrypt	1.0
RC6	Tinycrypt	1.0
RC6	Pravin-Nagare	1.0
RC6	shashankrao [1]	1.0
Anagrams	Custom	1.0
Hexprint	Custom	1.0

```
1 vagrant@Derp3:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/MyPinTool/majorkey/
      keyfinder_dumps$ ls
2 trace1.json   trace2.json
```

LISTING 5.8: Keyfinder Directory: ls command

```
1 vagrant@Derp3:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/MyPinTool/majorkey/
      keyfinder$ pypy __main__.py ../keyfinder_dumps/
2 ../keyfinder_dumps/trace1.json
3 ../keyfinder_dumps/trace2.json
4 Key is 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x61 0x62 0x63 0x64 0x65 0
      x66 0x30 0x31 0x31 0x32 0x32 0x33 0x33 0x34 0x34 0x35 0x35 0x36 0x36 0x37 0
      x37 0x38 0x38 0x39 0x39 0x61 0x61 0x62 0x62 0x63 0x63 0x64 0x64 0x65 0x65 0
      x66 0x66 0x30 0x31 0x30 0x33 0x32 0x35 0x34 0x37 0x36 0x39 0x38 0x62 0x61 0
      x64 0x63 0x66 0x65
5 123456789abcdef011223344556677889aabbccddeeff01032547698badcfe
```

LISTING 5.9: Keyfinder Tool Output

The directory `majorkey/keyfinder_dumps` now contain two files, one for each run, as depicted in Listing 5.8. We can then run the `keyfinder` tool and as we can see in Listing 5.9, the tool is correctly able to differentiate between the key and plaintexts. Although the use case of `keyfinder` is fairly trivial here as we could have carried out the analysis by inspection from looking at MajorKey's outputs for both traces. The `keyfinder` tool increases the reliability when more than one pair of potential keys and plaintexts are outputted by the tool. The run through of RC6 shows that MajorKey is able to successfully recover the plaintext, key and ciphertext from an application encrypting data. Furthermore it shows that our tool can differentiate between the key and different plaintexts.

5.3 Functionality and Performance

To evaluate our tool's ability to recover cryptographic data we analyse eleven different applications from seven libraries. Table 5.1 gives an overview of the applications tested in the following section, and Table 5.2 shows the parameters passed into the applications. We include analysis on an applications that do not carry out any cryptographic functions,

TABLE 5.2: Parameter Size Overview for Testing Applications

Algorithm	Implementation	Key size	Input size	Output size	Rounds
XOR	Custom	128 byte	4096 byte	4096 byte	NA
Obfuscated XOR	Custom	128 byte	4096 byte	4096 byte	NA
RC4-CBC	OpenSSL	128 bit	64 bit	64 bit	12
DES-CBC	OpenSSL	56 bits	64 bits	64 bits	16
DES-CBC	Tareque Hossain	56 bits	64 bits	64 bits	16
DES-CBC	Tinycrypt	56 bits	64 bits	64 bits	16
RC6	Tinycrypt	256 bits	128 bits	128 bits	20
RC6	Pravin-Nagare	256 bits	128 bits	128 bits	20
RC6	shashankrao	256 bits	128 bits	128 bits	20
Anagrams	Custom	NA	NA	NA	NA
Hexprint	Custom	NA	NA	NA	NA

TABLE 5.3: Filter Performance Results for Testing Applications

Algorithm	Implementation	Bitwise	Entropy	Read/Writes	Keyfinder
XOR	Custom	Yes	Yes	Yes	Yes
Obfuscated XOR	Custom	Yes	Yes	Yes	Yes
RC4-CBC	OpenSSL	Yes	Yes	Yes	Yes
DES-CBC	OpenSSL	Yes	Yes	No	Yes
DES-CBC	Tareque Hossain	Yes	Yes	No	Yes
DES-CBC	Tinycypt	Yes	Yes	No	Yes
RC6	Tinycrypt	Yes	No	Yes	Yes
RC6	Pravin-Nagare	Yes	Yes	Yes	Yes
RC6	shashankrao	Yes	Yes	Yes	Yes
Anagrams	Custom	No	No	No	No
Hexprint	Custom	No	No	No	No

mainly `Anagrams` that detects if two words are anagram of each other and `Hexprint` which takes in user input and prints the hexadecimal representation of it. We provide analysis of these two non cryptographic functions to show that we correctly identifying cryptographic loops and do not accept any program loop. For the rest of the applications tested, all of the inputs parameters are of standard size that would be expected for the respective cryptographic algorithm. Note that we tested `XOR` with two different input sizes 128 bytes and 4096 bytes as it allowed us to ensure we could detect the `XOR` operation regardless of size. Some of the algorithms with multiple different implementations to ensure that our tool can successfully detect cryptographic data across implementations. It allows us to evaluate how flexible our tool is and ensures it is not just able to provide results for specific library implementations of cryptographic functions. Furthermore, we have kept the inputs to each implementation constant so we do not have different input data affecting the run times or analysis.

For each of the applications tested, Table 5.3 shows which of our filters are able to detect the cryptographic data relevant to the application. Here we analyse each of the four filters we implemented against detection of the cryptographic data. The filters are, as discussed

in Section 3.6, the following:

1. **Bitwise:** Loops that contain a high percentage of bitwise and arithmetic instructions.
2. **Entropy:** Loops that have sufficient decreasing entropy between inputs and outputs.
3. **Read/Writes:** Loops that read from consecutive memory addresses in the first iteration, and write to consecutive memory addresses in the last iteration.
4. **Keyfinder:** Our key detection tool, which takes in *multiple* findings from different application runs and tries to reports back the encryption key being used.

We note that this table reports whether the filter detected the target data. During the run of our application, some false positives were generated by our filters, these are discussed further below and the results here do not show if a filter detected a false positive.

For all of the algorithms, the `bitwise` filter was able to detect the cryptographic loops. Furthermore, the `keyfinder` filter also successfully recovered full or partial keys for every application. The `entropy` filter was similarly effective, but less precise as it detected cryptographic data for all the tools except for `Tinycrypt RC6`. In order to remedy this, we tested various entropy threshold to see if a reduced decreased entropy threshold would allow us to detect the cryptographic data. We found that by lowering the threshold by 15% the `entropy` filter was able to detect the cryptographic data for `Tinycrypt RC6` but it had the side effect of increasing the number of false positives we saw in other algorithms, therefore we chose not to alter our threshold.

The `Read/Write` filter was effective with all applications except for `DES-CBC` as regardless of the library being used it detected a large number of false positives. We believe this could be due to a `DES-CBC` algorithm specific in how data is read and written during the encryption process as the problem is common to both implementations but more investigation is required. Most of the filters detect the cryptographic data in all the experiments. These results show that our tool achieves a high confidence of over 75%. As we see further in the evaluation potential cryptographic data that is only detected by one or two filters will have a lower confidence score and is usually indicative of a false positive.

We were able to recover the input text, key and output for all of the test applications except for the cryptographic output for `OpenSSL DES-CBC` and `shashankrao RC6`. Table 5.4 provides a breakdown of which cryptographic data our tool recovered for each testing application. Furthermore, for `RC6` across all three libraries we were able to recover round keys and intermediate cryptographic.

As mentioned above, our tool is correctly able to detect cryptographic data, but also generates false positives. Table 5.5 shows the number of false positives we saw for each application for each filter and the total represents the number of unique false positives, as some false positives were generated by more than one filter. Overall, there are very few false positives but there is a large number for `OpenSSL` implementations of functions and this appears across all four filters. The `Read/Writes` filter fared the worst with more than 10 false positives for both `RC4-CBC` and `DES-CBC`, and responsible for

TABLE 5.4: Cryptographic Material Detection Results for Testing Applications

Algorithm	Implementation	Input	Key	Output	Other
XOR	Custom	Yes	Yes	Yes	NA
Obfuscated XOR	Custom	Yes	Yes	Yes	NA
RC4-CBC	OpenSSL	Yes	Yes	Yes	Recovered round keys
DES-CBC	OpenSSL	Yes	Yes	No	NA
DES-CBC	Tareque Hossain	Yes	Yes	Yes	NA
DES-CBC	Tinycrypt	Yes	Yes	Yes	NA
RC6	Tinycrypt	Yes	Yes	Yes	Recovered round keys
RC6	Pravin-Nagare	Yes	Yes	Yes	Recovered round keys
RC6	shashankrao	Yes	Yes	No	Recovered 4 round keys
Anagrams	Custom	No	No	No	NA
Hexprint	Custom	No	No	No	NA

TABLE 5.5: False Positives in Cryptographic Data for Testing Applications

Algorithm	Implementation	Bitwise	Entropy	Read/Writes	Keyfinder	Total
XOR	Custom	0	0	0	0	0
Obf. XOR	Custom	0	0	0	0	0
RC4-CBC	OpenSSL	4	3	14	2	16
DES-CBC	OpenSSL	0	3	19	0	21
DES-CBC	Tareque Hossain	1	2	4	0	6
DES-CBC	Tinycrypt	1	2	3	0	6
RC6	Tinycrypt	0	0	0	0	0
RC6	Pravin-Nagare	0	1	0	0	1
RC6	shashankrao	0	2	7	5	11
Anagrams	Custom	0	0	0	0	0
Hexprint	Custom	0	0	0	0	0

TABLE 5.6: Speed Performance of our Instrumentation Tool

Algorithm	Implementation	Instrumentation time	Original time	Factor
XOR	Custom	855ms	3ms	285
Obfuscated XOR	Custom	825ms	2ms	412
RC4-CBC	OpenSSL	920ms	4ms	230
DES-CBC	OpenSSL	820ms	8ms	102
DES-CBC	Tareque Hossain	769ms	4ms	192
DES-CBC	Tinycrypt	800ms	5ms	160
RC6	Tinycrypt	1100ms	7ms	157
RC6	Pravin-Nagare	760ms	7ms	108
RC6	shashankrao	4830ms	47ms	102
Anagrams	Custom	791ms	3ms	263
Hexprint	Custom	922ms	5ms	198

over 70% of the false positives detected by our tool. We believe this is due to `OpenSSL` implementations having large overheads in its cryptographic functions resulting in more reads and writes from memory that our tool was went on to detect. However, as the false positives were normally only detected by one filter, they were displayed to the user with a lower confidence score which already suggests to the user they may not be accurate. Furthermore there was also a large number of false positives generated when evaluating `RC6 shashankrao`, this could also be explained by a large overhead, as it's a Python implementation therefore we have to instrument Python running the program, not a binary executable.

Overall our tool functions well across a range of different encryption algorithms implemented in different libraries. Furthermore, it correctly does not give back any results for the programs that contain no encryption functions.

5.3.1 Speed Performance

To evaluate the speed of our tool we analyse the times taken for our instrumentation tool to run the application, we then compare the added *instrumentation* time to the original running time of the application. Alongside that, as we deem speed to be an important goal of this project, we evaluate our tool's *analysis* time for each application with parallelisation and PyPy, and without. This allows us to test our parallelised parsing and analysis infrastructure as discussed in Chapter 4.5.3.

Table 5.6 shows the running time for each application with and without instrumentation along with the factor increase the testing applications took with instrumentation. It is important to keep application instrumentation times as close to original application running times because an application may change what it does if it takes longer than expected to run certain functions so we need instrumentation to be fast. On average we are able to keep the instrumentation time at roughly a factor of 150. The slowest factors were 412 and 285 for `XOR` and `Obfuscated XOR` respectively. We believe this is due to those programs being the simplest, as such they have fast initial running times the overhead of the

TABLE 5.7: Speed Performance of our Analysis Tool

Algorithm	Impl.	Instructions	Parallelised time	Original time	Speedup
XOR	Custom	154426	30s	2m 59s	6.0
Obf. XOR	Custom	159013	28s	2m 12s	5.1
RC4-CBC	OpenSSL	312476	2m 21s	4m 22s	1.9
DES-CBC	OpenSSL	441034	2m 34s	6m 42s	2.2
DES-CBC	Hossain	1180443	7m 21s	28m 34s	3.9
DES-CBC	Tinycrypt	401466	3m 24s	8m 54s	2.6
RC6	Tinycrypt	264141	36s	3m 21s	6.1
RC6	Nagare	354351	1m 50s	6m 02s	3.2
RC6	shashankrao	38179066	35m 50s	2hrs 4m 29s	3.5
Anagrams	Custom	129013	21s	2m 34s	7.5
Hexprint	Custom	141792	24s	2m 54s	7.3

instrumentation tool had a greater effect. This is further visible in the 263 factor increase for `Anagrams`.

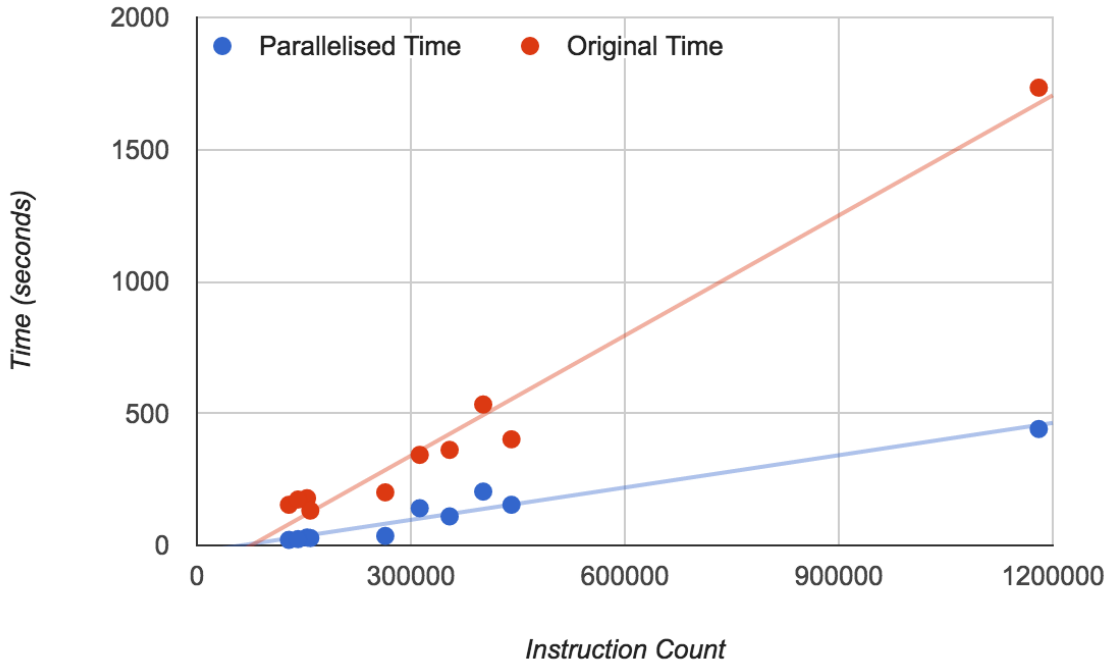

FIGURE 5.1: Speed Performance of our Analysis Tool

As the aapplications passed into MajorKey increase in size, the time taken to process them increases. Without the implementation of our parallelisation and speedup techniques the rate of increase with size is four times faster than without [2]. Overall our tool provided speedups between a factor of 1.9 and 6.1 [3], which shows that our switch to PyPy, and our use of `Multiprocessing` provided large speed ups. This can be seen in Table 5.7 and

[2] RC6 shashankrao has not been added to Figure 5.1 due to its size.
[3] This excludes `Anagrams` and `Hexprint` as they are not cryptographic programs.

TABLE 5.8: OpenSSL DES-CBC Comparison

OpenSSL DES-CBC	MajorKey	Grobert	Lutz
Plaintext	Recovered	Recovered	NA
Ciphertext	Recovered	Recovered	NA
Key	Recovered	Recovered	NA

TABLE 5.9: OpenSSL RC4-CBC Comparison

OpenSSL RC4-CBC	MajorKey	Grobert	Lutz
Plaintext	Recovered	Recovered	Recovered
Ciphertext	Recovered	Recovered	No
Key	Recovered	Partial	No

Figure 5.1 show the analysis time taken by MajorKey before we added parallelisation and PyPy to our tool, and afterwards.

5.4 Comparison with Other Tools

In Section 2.3, we discussed related works. There are four main related works that tackle the same or a similar problem to our work, *recovering cryptographic data from binaries*. It is hard to directly compare our results against the other tools as we do not have access to the source code and therefore have to rely on what they say in their papers however we cannot guarantee results as they are not reproducable. Furthermore, as we test our application against different encryption algorithms, a direct comparison is not possible.

In the rest of this section our work is evaluated against works by Grobert [12] and Lutz [18]. Note that comparison with Lutz's work is hard as the instrumentation traces generated by his tool has been created using taint analysis, whereas ours has not.

TABLE 5.10: XOR Comparison

XOR	MajorKey	Grobert	Lutz
Plaintext	Recovered	Recovered	NA
Ciphertext	Recovered	Recovered	NA
Key	Recovered	Recovered	NA

TABLE 5.11: Speed Performance Comparison

Algorithm	MajorKey time	Grobert time	Lutz time
XOR	30s	1m 42s	NA
DES	2m 34s	3m 3s	46s
RC4	2m 21s	13m 44s	NA

5.4.1 Results

Tables 5.8, 5.9 and 5.10 above show how our tool compared to the respective other works. Where an `NA` is present it mean that we could not test against it because the report provided insufficient evaluation against the given test application. Our tool, and Grobert's tool have the advantage of being able to recover ciphertexts and well as plaintexts, Lutzs does not. If the application being analysed by our respective tools receives a large stream of encrypted text, ours are more likely to recover the cryptographic data as it can be detected without any prior knowledge, whereas Lutz taints the ciphertext being passed into the tool.

Our tool succesfully recovers data for `DES`, `RC4` and `XOR`. Lutz's tool was only able to recover the plaintext for `RC4` which means our tool is stronger due to it's ability to detect the key and ciphertext. Furthermore, as can be seen in Table 5.9 our tool is able to recover the full key for `RC4` whereas Grobert's was only able to recover a partial key. Overall a direct comparison is hard as the intersecting test beds are limited but from the results our tool has fared the same if not better than the two compared works.

5.4.2 Speed Performance

We compare speed performance for the same algorithms as above, `DES`, `RC4` and `XOR`. Table 5.11 shows the times taken for instrumentation and analysis of each application where `NA` indicates insufficient data to carry out the evaluation. Our tool is on average twice as fast as Grobert's which is a success. This is a positive result as our tool is also most similar to Grobert's as we instrument traces in the same manner. However, Lutz's tool appears to be roughly three times faster than ours for the `DES` result, this may be due to Lutz's trace tool using taint analysis causing it to produce a much smaller trace, further analysis is needed though.

5.5 Case Study: RC6 Chat Application

We now demonstrate MajorKey with a larger, more complex application. The previous testing applications have been library implementations that took in an input, encrypted or decrypted it with respect to the cryptographic algorithm in question and outputted the results. Here we present our tool how our tool works when given a chat application to analyse.

5.5.1 Chat Application Overview

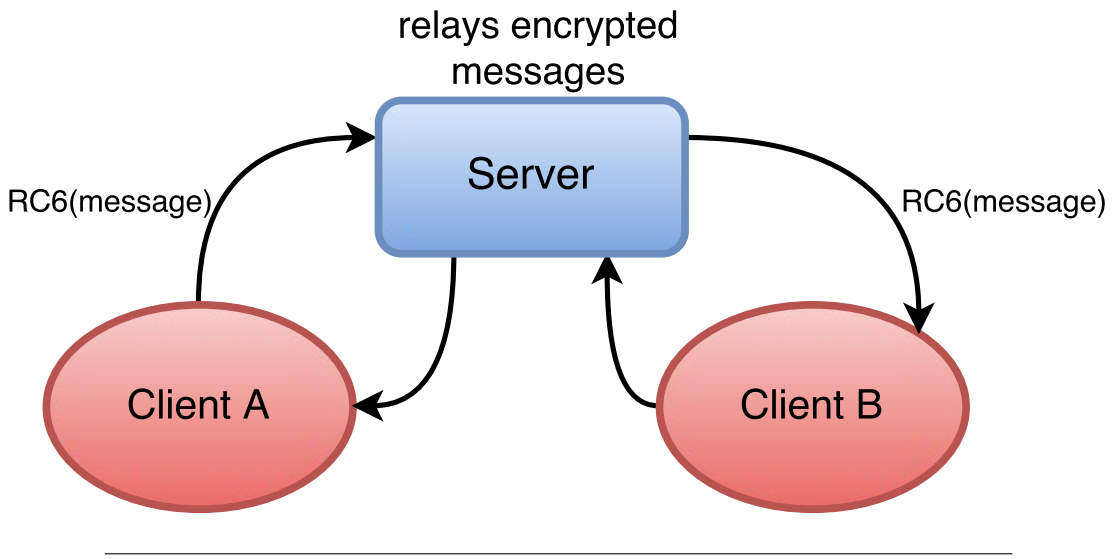

FIGURE 5.2: RC6 Chat Application Setup

The set up of our application can be seen in Figure 5.2. `Client A` can send messages to `Client B` via the server. An example message flow would be the following:

1. User sends message using `Client A`.
2. `Client A` then encrypts the message using `RC6`.
3. The message is then sent by `Client A` to the server.
4. The server forwards the encrypted message to `Client B`.
5. `Client B` decrypts the message.
6. User can now read the original message sent.

Our chat application is a command line program written in Python using Sockets and `shashankrao`'s implementation of `RC6`. For testing, as shown in **AppendixD**, we run a server locally on port 5000, and open two clients in different consoles which can pass messages using the local server.

We test our application with three plaintext messages sent from `Client A` to `Client B`, *being instrumented*, using the same key for all three messages as shown in Listing 5.10.

```
Message 1:
'Hii how are you?'
Message 2:
'I am very sleepy'
Message 3:
'Pls respond, why you no respond to me?!'

Key:
'A WORD IS A WORD'
```

LISTING 5.10: RC6 Chat Application Inputs and Key

5.5.2 Generating an Instrumentation Trace

As mentioned above, there are three main parts to our chat application, the two clients and the server. Before we can generate a trace, we must decide what we will instrument. We instrument one of the two clients as we are interested in recovering the cryptographic data. The server just relays the data, whereas the applications have the ability to encrypt and decrypt messages, therefore through instrumenting clients we will be able to recover the inputs, and output to the encryption.

```
vagrant@Derp3:~/majorkey$ ./../../../pin.sh -t obj-intel64/MyPinTool.so -istart
    15000 -- python ~/examples/pyrc6/client.py localhost 5000
```

LISTING 5.11: RC6 Chat Application Instrumentation Trace Command

```
vagrant@Derp3:~/majorkey$ wc -l trace.out
4143136
```

LISTING 5.12: RC6 Chat Application Initial Instrumentation Trace Line Count

To generate our trace, we run `Client B` inside of our Pintool using the command shown in Listing 5.11 which starts a chat application client listing on port 5000. As can been seen in Listing 5.12, running `word count` to report the number of lines in the output trace file shows if we begin instrumentation straight away will contain over 4,000,000 instructions by the time one message has been received. Not only does it take time to generate this trace as each of the 4,000,000 instructions have to be instrumented but it will take over 3 hours to analyse.

Consequentially we pass a starting instruction count, as described in 4.2, to skip over the unnecessary instructions at the start. This not only increases the speed of instrumentation but reduces the trace size. Unnecessary instructions were determined to be ones that were ran before the application had received it's first message. To assess where this boundary was, we ran the application five times and before `Client A` sent a message we retrieved the number of lines in the trace file. Then we took the lowest number [4] reported by the five runs as the number of instructions to skip at the beginning of instrumentation. As the application will not have received any messages at that point, we can assume all the instructions called before are not relevant to the recovery of cryptographic material as no encrypted data has been sent to the tool yet, and therefore nothing no functions of interest to our tool will have been called.

5.5.3 Running MajorKey 's Analysis Tool

Using the above method to reduce trace size, instead of generating a trace with over four million instructions we were able to reduce that number to just over a million, a 75% trace size reduction.

```
vagrant@Derp3:~/majorkey$ pypy majorkey ../trace.out --keyfinder='
    keyfinder_dumps'
```

LISTING 5.13: RC6 Chat Application MajorKey Command

[4] The lowest number was chosen as we did not want to miss any instructions after trace.

This instruction trace is then passed to our analysis tool using the instruction in Listing 5.13. The `keyfinder` argument is passed in to ensure our results are saved in a `JSON` file for further analysis with other messages.

```
vagrant@Derp3:~/majorkey$ pypy majorkey ../trace_one.out --keyfinder='
    keyfinder_dumps'

Confidence: 1.0
--------------------------------------------------
potential inputs
0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20 0x41 0x20 0x57 0x4f 0x52 0x44
     0x44
A WORD IS A WORDD
0x48 0x69 0x69 0x20 0x68 0x6f 0x77 0x20 0x61 0x72 0x65 0x20 0x79 0x6f 0x75 0x3f
Hii how are you?
potential outputs
0xda 0x6d 0xca 0xa6 0xdf 0x43 0x0e 0x2b 0x27 0xd6 0x26 0x94 0xe4 0x56 0x96 0x49
--------------------------------------------------
Confidence: 0.5
--------------------------------------------------
potential inputs
0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20 0x41 0x20 0x57 0x4f 0x52 0x44
     0x44
A WORD IS A WORDD
0x20 0x63 0x42 0x12 0x42 0x42 0x42 0x12 0x61 0x49 0x65 0x80 0x6a 0x71 0x33 0x33
potential outputs
0xda 0x6d 0xca 0xa6 0xdf 0x43 0x0e 0x2b 0x27 0xd6 0x26 0x94 0xe4 0x56 0x96 0x49
--------------------------------------------------
...
```

LISTING 5.14: Output for Message 1

The output of our tool for the trace generated by the first message, *line 2 in Listing 5.10* is shown in Listing 5.14 and the subsequent outputs for messages two and three, lines 4 and 6 of the listing respectively are shown in **AppendixE**. As can be seen, all three inputs, key and respectively outputs are recovered by MajorKey along with false positives for Message 1 and Message 2. To increase our confidence, and consequentially reduce the number of false positives, and distinguish between the key and plaintexts we pass our results into `keyfinder`.

```
vagrant@Derp3:~/majorkey$ pypy keyfinder keyfinder_dumps
Key is 0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20 0x41 0x20 0x57 0x4f 0
    x52 0x44 0x44
A WORD IS A WORDD
Messages with a confidence of 1 are:
0x48 0x69 0x69 0x20 0x68 0x6f 0x77 0x20 0x61 0x72 0x65 0x20 0x79 0x6f 0x75 0x3f
Hii how are you?
0x49 0x20 0x61 0x6d 0x20 0x76 0x65 0x72 0x79 0x20 0x73 0x6c 0x65 0x65 0x70 0x79
I am very sleepy
0x50 0x6c 0x73 0x20 0x72 0x65 0x73 0x70 0x6f 0x6e 0x64 0x2c 0x20 0x77 0x68 0x20
     0x79 0x20 0x79 0x6f 0x75 0x20 0x6e 0x6f 0x20 0x72 0x65 0x73 0x70 0x6f 0x6e
     0x20 0x64 0x20 0x74 0x6f 0x20 0x6d 0x65 0x3f 0x21 0x20 0x20 0x20 0x20 0x20
     0x20 0x20
Pls respond, wh y you no respon d to me?!
Messages with a confidence of 0.5 are:
0x20 0x63 0x42 0x12 0x42 0x42 0x42 0x12 0x61 0x49 0x65 0x80 0x6a 0x71 0x33 0x33
0x73 0x68 0x63 0x68 0x6e 0x61 0x6c 0x6a 0x53 0x68 0x68 0x41 0x73 0x21 0x52 0x44
shchnaljShhAs!RD
```

LISTING 5.15: Keyfinder Outputs

`Keyfinder` is run with the path to the directory containing the `JSON` dumps our tool generated when analysing the traces for the encrypted messages received by `Client B`. Analysis of `keyfinder` is shown in Listing 5.15, here we can see on line 3 it has correctly recovered the key, and the three plaintexts with a confidence of one as shown on lines 6, 8 and 10 respectively. Furthermore it outputs two potential messages with half the confidence on lines 12 and 13 respectively. These two messages are not false positives but have been identified as they were found in cryptographic loops that use the some of the key. E.G. The input message `shchnaljShhAs!RD` as shown in Listing E.1 is found with in a potentially cryptographic loop along with the partial key `A WORD IS`. Nevertheless, as you can see all data was found.

5.5.4 Conclusion

MajorKey successfully recovered both the inputs, key and outputs passed to our chat application. We consider this a success as it was able to do so despite a variety of other application processing for receiving and sending messages to the server happening at the same time.

The set up described in this section is typical of applications that exist today that are connected to the Internet of Things. A real word scenario representative of our case study could be a user sending a message between an application on their phone, `Client A` in 5.2, and the heating system in their house `Client B`, via a server owned by the company that designed their smart heating system, the server. The successful analysis of this case study shows the strength of MajorKey and it's ability to help security researchers analyse and secure existing applications and technologies used by people every day.

Chapter 6

Conclusion

MajorKey is an end-to-end suite of tools to enable recovery of cryptographic data from applications using encryption. To achieve this goal we have developed a cross platform instrumentation tool which is able to generate traces from the run of an executable. Alongside that, we produced a command line tool that automatically detect cryptographic operations within the trace generated for the run of an application. Once cryptographic operations have been detected, we are then able to recover the input and the key to that function as well as the output. Furthermore, we detect intermediate round keys and ciphertexts when block ciphers are used.

6.1 Contributions

The following section summarises our key achievements.

MajorKey Instrumentation Tool

Our tool can take in any binary, analyse it and generate traces with a reasonable slowdown. As application size increases, the efficiency of our tool does as the initial instrumentation setup overhead is spread across a longer run. Our instrumentation tool outputs results in a specific message format designed for our command line analysis tool. Furthermore, to help reduce the trace size outputted by the tool, it accepts a start memory address, end memory address and function name as arguments to control what parts of the executable is instrumented - this also has the added benefit of decreasing the runtime of the tool.

MajorKey Python Tool

We developed a Python framework that allows for detection of cryptographic loops and recover cryptographic data, and a command line tool that implement the said framework. Our framework has been designed with speed in mind, and is on average twice as fast as the most similar existing tool. It can be easily extended to work with new filters and be built on top of. Moreover it is flexible and able to accept traces created using a range of different instrumentation tools.

Recovery of Cryptographic Material

MajorKey is able to recover cryptographic materials, including inputs, outputs, keys and intermediate results from XOR, RC4, DES and RC6. Furthermore, although our tool is not designed to work with obfuscated binaries, we are able to successfully analyse a simple obfuscated XOR application.

RC6 Evaluation

Our evaluation against RC6 represents a novel contribution as it has not been successfully evaluated before. We tested against two different library implementations and got successful results for recovering the input, key, output and intermediate round keys and ciphertexts for both. RC6 is a secure encryption algorithm that has wide usage today, including being used in malware distributed by the NSA [29].

RC6 Chat Application Case Study

We successfully evaluated gainst a complex application receiving messages via sockets from another client. The key, inputs and outputs were recovered from messages sent using the RC6 encryption algorithm. The data communication application tested in this thesis is in line with real world applications proving our tool is capable of handling programs of such scale.

MajorKey Keyfinder Filter

A new filtering technique that is able to analyse multiple traces to make inferences about the key and plaintext. This method works when the key used is the same across the traces being compared, and this is normally the case with symmetric ciphers as they are designed to be long term. Keyfinder increases our confidence in the key and plaintext pair, and can help us determine the correct key plaintext pair if our main tool outputs multiple potential key and plaintext pairs. This method worked successfully for all of the applications we tested.

6.2 Future Work

This section discusses various improvements that could be made to MajorKey in the future. A majority of the items mentioned we would like to have implemented or tried had more time been available.

Obfuscated Binaries

Extending our tool to work with obfuscated binaries is a future work that can be undertaken. This is a large task due to the different methods that are employed to obfuscate binaries. One of which would be embedding unnecessary arithmetic and bitwise loops in the normal application processing code to throw off MajorKey's filters. How to prevent an application evading our analysis through those means still remains a technical challenge.

Multiple Encryption Applications

MajorKey has only been tested on applications that implement one cryptographic function e.g. a DES encryption or a RC6 encryption. It has not been tested against applications which wrap data in multiple layers of encryption. Although one can argue this is unlikely to occur in every day applications, if we were to analyse botnets or other forms of malware it may appear and would be an interesting next step.

Reevaluating OpenSSL

As shown in the evaluation, our tool struggles in recovering data from OpenSSL implementations of algorithms. We would like to investigate into this issue further to establish why this occurs. Due to a large number of false positives being detected we believe that the problem could be due to the thresholds set for our cryptographic loop detection filters. If this is the case then one solution would be to manually override them at runtime.

Ransomware Testing

Currently most of our analysis has been carried out on simple applications implementing cryptographic functions, with more time we would like to investigate how MajorKey performs when given a larger application such as randsomware. As ransomware traces tend to be large, often 5GB upwards, and more complex program they will truly test MajorKey's ability.

Appendix A

DES Objdump output

```
vagrant@Derp3:~/examples$ objdump -t DES/run_des.o

DES/run_des.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000400238 l    d  .interp  0000000000000000              .interp
0000000000400254 l    d  .note.ABI-tag  0000000000000000              .note.ABI-tag
0000000000400274 l    d  .note.gnu.build-id 0000000000000000              .note.gnu.build-id
0000000000400298 l    d  .gnu.hash  0000000000000000              .gnu.hash
00000000004002b8 l    d  .dynsym  0000000000000000              .dynsym
0000000000400498 l    d  .dynstr  0000000000000000              .dynstr
000000000040055c l    d  .gnu.version 0000000000000000              .gnu.version
0000000000400588 l    d  .gnu.version_r 0000000000000000              .gnu.version_r
00000000004005c8 l    d  .rela.dyn  0000000000000000              .rela.dyn
00000000004005e0 l    d  .rela.plt  0000000000000000              .rela.plt
00000000004007a8 l    d  .init  0000000000000000              .init
00000000004007d0 l    d  .plt 0000000000000000              .plt
0000000000400910 l    d  .text  0000000000000000              .text
0000000000401b54 l    d  .fini  0000000000000000              .fini
0000000000401b60 l    d  .rodata  0000000000000000              .rodata
0000000000401e18 l    d  .eh_frame_hdr  0000000000000000              .eh_frame_hdr
0000000000401e78 l    d  .eh_frame  0000000000000000              .eh_frame
0000000000602e10 l    d  .init_array  0000000000000000              .init_array
0000000000602e18 l    d  .fini_array  0000000000000000              .fini_array
0000000000602e20 l    d  .jcr 0000000000000000              .jcr
0000000000602e28 l    d  .dynamic 0000000000000000              .dynamic
0000000000602ff8 l    d  .got 0000000000000000              .got
0000000000603000 l    d  .got.plt 0000000000000000              .got.plt
00000000006030c0 l    d  .data  0000000000000000              .data
0000000000603e20 l    d  .bss 0000000000000000              .bss
0000000000000000 l    d  .comment 0000000000000000              .comment
0000000000000000 l    df *ABS*  0000000000000000              run_des.c
0000000000603e50 l     O .bss 0000000000000008              key_file
0000000000603e40 l     O .bss 0000000000000008              input_file
0000000000603e30 l     O .bss 0000000000000008              output_file
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
0000000000602e20 l     O .jcr 0000000000000000              __JCR_LIST__
0000000000400e80 l     F .text  0000000000000000              deregister_tm_clones
0000000000400eb0 l     F .text  0000000000000000              register_tm_clones
```

```
0000000000400ef0 l     F .text  0000000000000000              __do_global_dtors_aux
0000000000603e20 l     O .bss 0000000000000001              completed.6973
0000000000602e18 l     O .fini_array  0000000000000000              __do_global_dtors_aux_fini_array_entry
0000000000400f10 l     F .text  0000000000000000              frame_dummy
0000000000602e10 l     O .init_array  0000000000000000              __frame_dummy_init_array_entry
0000000000000000 l    df *ABS*  0000000000000000              des.c
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
00000000004020d0 l     O .eh_frame  0000000000000000              __FRAME_END__
0000000000602e20 l     O .jcr 0000000000000000              __JCR_END__
0000000000000000 l    df *ABS*  0000000000000000
0000000000602e18 l       .init_array  0000000000000000              __init_array_end
0000000000602e28 l     O .dynamic 0000000000000000              _DYNAMIC
0000000000602e10 l       .init_array  0000000000000000              __init_array_start
0000000000603000 l     O .got.plt 0000000000000000              _GLOBAL_OFFSET_TABLE_
0000000000401b50 g     F .text  0000000000000002              __libc_csu_fini
0000000000000000       F *UND*  0000000000000000              free@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              putchar@@GLIBC_2.2.5
0000000000603860 g     O .data  0000000000000100              S2
0000000000603260 g     O .data  0000000000000100              S8
0000000000603460 g     O .data  0000000000000100              S6
0000000000603660 g     O .data  0000000000000100              S4
0000000000000000  w      *UND*  0000000000000000              _ITM_deregisterTMCloneTable
00000000006030c0  w      .data  0000000000000000              data_start
0000000000000000       F *UND*  0000000000000000              puts@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              fread@@GLIBC_2.2.5
00000000004014a0 g     F .text  000000000000063c              process_message
0000000000603e20 g       .data  0000000000000000              _edata
0000000000000000       F *UND*  0000000000000000              clock@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              fclose@@GLIBC_2.2.5
0000000000401b54 g     F .fini  0000000000000000              _fini
0000000000000000       F *UND*  0000000000000000              __stack_chk_fail@@GLIBC_2.4
0000000000603a60 g     O .data  00000000000000c0              message_expansion
0000000000000000       F *UND*  0000000000000000              memset@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              __libc_start_main@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              srand@@GLIBC_2.2.5
00000000006030c0 g       .data  0000000000000000              __data_start
0000000000603d40 g     O .data  00000000000000e0              initial_key_permutaion
0000000000000000       F *UND*  0000000000000000              ftell@@GLIBC_2.2.5
0000000000000000  w      *UND*  0000000000000000              __gmon_start__
00000000006030c8 g     O .data  0000000000000000              .hidden __dso_handle
0000000000400fe0 g     F .text  00000000000001ba              print_key_set
0000000000401b60 g     O .rodata  0000000000000004              _IO_stdin_used
0000000000000000       F *UND*  0000000000000000              time@@GLIBC_2.2.5
0000000000603960 g     O .data  0000000000000100              S1
```

```
84 00000000006031e0 g     O .data  0000000000000080
       right_sub_message_permutation
85 0000000000400f90 g     F .text  000000000000004d              generate_key
86 0000000000603360 g     O .data  0000000000000100              S7
87 0000000000603560 g     O .data  0000000000000100              S5
88 0000000000401ae0 g     F .text  0000000000000065              __libc_csu_init
89 0000000000603760 g     O .data  0000000000000100              S3
90 0000000000000000       F *UND*  0000000000000000              malloc@@GLIBC_2
       .2.5
91 0000000000603e58 g       .bss 0000000000000000              _end
92 0000000000400e4c g     F .text  0000000000000000              _start
93 0000000000000000       F *UND*  0000000000000000              fseek@@GLIBC_2
       .2.5
94 00000000006030e0 g     O .data  0000000000000100
       final_message_permutation
95 0000000000603e20 g       .bss 0000000000000000              __bss_start
96 0000000000400910 g     F .text  000000000000053c              main
97 0000000000000000       F *UND*  0000000000000000
       __printf_chk@@GLIBC_2.3.4
98 00000000004011a0 g     F .text  00000000000002f9              generate_sub_keys
99 0000000000000000       F *UND*  0000000000000000              fopen@@GLIBC_2
       .2.5
100 0000000000000000  w      *UND*  0000000000000000
       _Jv_RegisterClasses
101 0000000000603c40 g     O .data  0000000000000100
       initial_message_permutation
102 0000000000603b20 g     O .data  00000000000000c0
       sub_key_permutation
103 0000000000000000       F *UND*  0000000000000000              fwrite@@GLIBC_2
       .2.5
104 0000000000603e20 g     O .data  0000000000000000              .hidden
       __TMC_END__
105 0000000000000000  w      *UND*  0000000000000000
       _ITM_registerTMCloneTable
106 0000000000603be0 g     O .data  0000000000000044              key_shift_sizes
107 0000000000400f40 g     F .text  000000000000004c
       print_char_as_binary
108 00000000004007a8 g     F .init  0000000000000000              _init
109 0000000000000000       F *UND*  0000000000000000              rand@@GLIBC_2.2.5
```

Appendix B

RC6 Control flow graph

FIGURE B.1: Overview of entire CFG

FIGURE B.2: Close up of CFG nodes

Appendix C

Intermediate RC6 rounds

```
1 ['0x7ffe0b8cb020', '0x7ffe0b8cb01c', '0x7ffe0b8cb018', '0x7ffe0b8cb014', '0
    x7ffe0b8cb010', '0x7ffe0b8cb00c', '0x7ffe0b8cb008', '0x7ffe0b8cb004', '0
    x7ffe0b8cb000', '0x7ffe0b8caffc', '0x7ffe0b8caff8', '0x7ffe0b8caff4', '0
    x7ffe0b8caff0', '0x7ffe0b8cafec', '0x7ffe0b8cafe8', '0x7ffe0b8cafe4', '0
    x7ffe0b8cafe0', '0x7ffe0b8cafdc', '0x7ffe0b8cafd8', '0x7ffe0b8cafd4', '0
    x7ffe0b8cafd0', '0x7ffe0b8cafcc', '0x7ffe0b8cafc8', '0x7ffe0b8cafc4', '0
    x7ffe0b8cafc0', '0x7ffe0b8cafbc', '0x7ffe0b8cafb8', '0x7ffe0b8cafb4', '0
    x7ffe0b8cafb0', '0x7ffe0b8cafac', '0x7ffe0b8cafa8', '0x7ffe0b8cafa4', '0
    x7ffe0b8cafa0', '0x7ffe0b8caf9c', '0x7ffe0b8caf98', '0x7ffe0b8caf94', '0
    x7ffe0b8caf90', '0x7ffe0b8caf8c', '0x7ffe0b8caf88', '0x7ffe0b8caf84']
2 ['0xfa90d1ed', '0xad452bb3', '0xa58d40b6', '0x7e657009', '0xd526156c', '0
    x97cb7f6a', '0x5b05e64a', '0x44e6348c', '0x3311b59e', '0x9f83bf9b', '0
    x7321e9a0', '0xe47ba777', '0x9134331', '0x343dd646', '0xb50191d6', '0
    xb10e0f99', '0xd944f68', '0xc8346538', '0xb7dbb70a', '0x1fd53a4c', '0
    xea48ed81', '0xfa93fe7b', '0xe11395bd', '0x3482f3a8', '0x3276050e', '0
    xf125a41f', '0x3023742c', '0x51849bbc', '0x57dffb86', '0x3f6096d', '0
    x73f747eb', '0xedeba31e', '0xfe4244d8', '0xe9ba8b67', '0x1c924934', '0
    x9cabadc', '0x1e729f9a', '0x6b7457a', '0xa8a83848', '0x35241302']
3 ['0xafbcdbd3', '0x59005c7d', '0x430caa83', '0xad452bb3', '0xdd355af7', '0
    x7e657009', '0x6ff2458c', '0x97cb7f6a', '0xf7e8c0c', '0x44e6348c', '0
    x2da353ec', '0x9f83bf9b', '0x5f934c5f', '0xe47ba777', '0xa215b2ca', '0
    x343dd646', '0x2322c9f9', '0xb10e0f99', '0x220a8b86', '0xc8346538', '0
    xf6b36779', '0x1fd53a4c', '0x39612c4', '0xfa93fe7b', '0xdabf139f', '0
    x3482f3a8', '0x37482596', '0xf125a41f', '0xdfd1a127', '0x51849bbc', '0
    x99765bd9', '0x3f6096d', '0x8dda4fbf', '0xedeba31e', '0x6002659b', '0
    xe9ba8b67', '0x5e84356b', '0x9cabadc', '0xbdac9b8a', '0x6b7457a']
4 ['0xad452bb3', '0xfef4d10b', '0x7e657009', '0xafbcdbd3', '0x97cb7f6a', '0
    x430caa83', '0x44e6348c', '0xdd355af7', '0x9f83bf9b', '0x6ff2458c', '0
    xe47ba777', '0xf7e8c0c', '0x343dd646', '0x2da353ec', '0xb10e0f99', '0
    x5f934c5f', '0xc8346538', '0xa215b2ca', '0x1fd53a4c', '0x2322c9f9', '0
    xfa93fe7b', '0x220a8b86', '0x3482f3a8', '0xf6b36779', '0xf125a41f', '0
    x39612c4', '0x51849bbc', '0xdabf139f', '0x3f6096d', '0x37482596', '0
    xedeba31e', '0xdfd1a127', '0xe9ba8b67', '0x99765bd9', '0x9cabadc', '0
    x8dda4fbf', '0x6b7457a', '0x6002659b', '0x35241302', '0x5e84356b']
5 ['0x6002659b', '0x6b7457a', '0x8dda4fbf', '0x9cabadc', '0x99765bd9', '0
    xe9ba8b67', '0xdfd1a127', '0xedeba31e', '0x37482596', '0x3f6096d', '0
    xdabf139f', '0x51849bbc', '0x39612c4', '0xf125a41f', '0xf6b36779', '0
    x3482f3a8', '0x220a8b86', '0xfa93fe7b', '0x2322c9f9', '0x1fd53a4c', '0
    xa215b2ca', '0xc8346538', '0x5f934c5f', '0xb10e0f99', '0x2da353ec', '0
    x343dd646', '0xf7e8c0c', '0xe47ba777', '0x6ff2458c', '0x9f83bf9b', '0
    xdd355af7', '0x44e6348c', '0x430caa83', '0x97cb7f6a', '0xafbcdbd3', '0
    x7e657009', '0xfef4d10b', '0xad452bb3', '0x485d4e67', '0x59005c7d']
```

```
6 ['0x7ffe0b8caf8c', '0x7ffe0b8caf90', '0x7ffe0b8caf94', '0x7ffe0b8caf98', '0
      x7ffe0b8caf9c', '0x7ffe0b8cafa0', '0x7ffe0b8cafa4', '0x7ffe0b8cafa8', '0
      x7ffe0b8cafac', '0x7ffe0b8cafb0', '0x7ffe0b8cafb4', '0x7ffe0b8cafb8', '0
      x7ffe0b8cafbc', '0x7ffe0b8cafc0', '0x7ffe0b8cafc4', '0x7ffe0b8cafc8', '0
      x7ffe0b8cafcc', '0x7ffe0b8cafd0', '0x7ffe0b8cafd4', '0x7ffe0b8cafd8', '0
      x7ffe0b8cafdc', '0x7ffe0b8cafe0', '0x7ffe0b8cafe4', '0x7ffe0b8cafe8', '0
      x7ffe0b8cafec', '0x7ffe0b8caff0', '0x7ffe0b8caff4', '0x7ffe0b8caff8', '0
      x7ffe0b8caffc', '0x7ffe0b8cb000', '0x7ffe0b8cb004', '0x7ffe0b8cb008', '0
      x7ffe0b8cb00c', '0x7ffe0b8cb010', '0x7ffe0b8cb014', '0x7ffe0b8cb018', '0
      x7ffe0b8cb01c', '0x7ffe0b8cb020', '0x7ffe0b8cb024', '0x7ffe0b8cb028']
7 ['0xa8a83848', '0x6002659b', '0x1e729f9a', '0x8dda4fbf', '0x1c924934', '0
      x99765bd9', '0xfe4244d8', '0xdfd1a127', '0x73f747eb', '0x37482596', '0
      x57dffb86', '0xdabf139f', '0x3023742c', '0x39612c4', '0x3276050e', '0
      xf6b36779', '0xe11395bd', '0x220a8b86', '0xea48ed81', '0x2322c9f9', '0
      xb7dbb70a', '0xa215b2ca', '0xd944f68', '0x5f934c5f', '0xb50191d6', '0
      x2da353ec', '0x9134331', '0xf7e8c0c', '0x7321e9a0', '0x6ff2458c', '0
      x3311b59e', '0xdd355af7', '0x5b05e64a', '0x430caa83', '0xd526156c', '0
      xafbcdbd3', '0xa58d40b6', '0xfef4d10b', '0xfa90d1ed', '0x485d4e67']
8 ['0x9cabadc', '0x5e84356b', '0xe9ba8b67', '0x6002659b', '0xedeba31e', '0
      x8dda4fbf', '0x3f6096d', '0x99765bd9', '0x51849bbc', '0xdfd1a127', '0
      xf125a41f', '0x37482596', '0x3482f3a8', '0xdabf139f', '0xfa93fe7b', '0
      x39612c4', '0x1fd53a4c', '0xf6b36779', '0xc8346538', '0x220a8b86', '0
      xb10e0f99', '0x2322c9f9', '0x343dd646', '0xa215b2ca', '0xe47ba777', '0
      x5f934c5f', '0x9f83bf9b', '0x2da353ec', '0x44e6348c', '0xf7e8c0c', '0
      x97cb7f6a', '0x6ff2458c', '0x7e657009', '0xdd355af7', '0xad452bb3', '0
      x430caa83', '0x59005c7d', '0xafbcdbd3', '0x89e4d7f0', '0xfef4d10b']
9 ['0x5c276548', '0xf520fa94', '0x4292f8b5']
10 ['0xb74df56e', '0x5c46a9ac', '0x1d572b13']
11 ['0x2056782f', '0x87a2e9ac', '0x19d36d38']
12 ['0xbdf307cb', '0x4fb3e6f2', '0xb267cfd0']
13 ['0x4a0044ae', '0x79d34ba1', '0x9cc9a909']
14 ['0x18a30af6', '0x47224421', '0x60cfb203']
15 ['0x62ce8ad5', '0x4396508f', '0x9c6787aa']
16 ['0xb1e5c9ff', '0x5cd76e9f', '0xcae80986']
17 ['0xbbf5d98a', '0x9748788b', '0x24767984']
18 ['0x80e1e1db', '0xbb675781', '0xee57ded7']
19 ['0xbc99e6b7', '0x405c5384', '0xe5ffbcf4']
20 ['0xd7a12765', '0x6d7bb7ca', '0x270960fc']
21 ['0xe9a4ce28', '0x5e46512f', '0x7c74a515']
22 ['0xba116f2', '0xa6503f61', '0x5f034ef9']
23 ['0xe092b680', '0x85695f15', '0x43c8ee38']
24 ['0xadd0e8c3', '0x3aaab847', '0x6cd53e60']
25 ['0xe4cd1bd0', '0x73b485e9', '0x1ac43630']
26 ['0xee3fb316', '0xbcdbf434', '0x249e9335']
27 ['0x4b4a9985', '0x17e21ec3', '0x995e8fa7']
28 ['0x9f0ab44a', '0xb6eaee2e', '0xc00a874a']
29 ['0xf93bb6f0', '0xee5973b2', '0x9ed327da']
30 ['0x73bfde4d', '0xc8faddb2', '0x5e043e95']
31 ['0x180b96dd', '0xa97cdabb', '0x7922e9b0']
32 ['0xf361dea6', '0xdc239031', '0xa1ee2334']
33 ['0x79715882', '0x8e1f2167', '0xa8bf3362']
34 ['0x32bd9751', '0xe07f649', '0x4ce52d93']
35 ['0x63edc9f', '0xfedfdd90', '0xa221dc15']
36 ['0x7594df90', '0xf7bdc219', '0xc252d591']
37 ['0xe0652e31', '0x89c99a69', '0xd5ad7ce2']
38 ['0xa1eb50d', '0x272bc443', '0xb65035df']
39 ['0xdb70333', '0x2ec62d6d', '0xbd17c84b']
40 ['0xef925212', '0xb8bdc9fb', '0xef8ad17d']
41 ['0x3de84e1e', '0x63dd1135', '0xcac74d45']
42 ['0x3efc0d8d', '0xa56d9b06', '0x1e2b3e22']
43 ['0x957231bf', '0x26d21c6b', '0x3e0a1a13']
44 ['0xcf6a5fe6', '0xdcf9832', '0x6ca3559d']
45 ['0xdfae2b05', '0x23724515', '0x158f86']
```

```
['0xbf0a8b1d', '0x170b693f', '0x8d4eee34']
['0xdbf2c6d6', '0xd32539e7', '0xab9b84c5']
['0x8b2045d9', '0xa74ed888', '0xf65f68ba']
['0xb5719c02', '0xe414e5de', '0xc6cd835e']
['0xd98b251b', '0x7af7b2ec', '0xf9422a66']
['0x76b7a6e7', '0xd08f21c5', '0x120160d7']
['0x3e4747ac', '0x80917a37', '0x21cb2309']
```

Appendix D

Case Study: RC6 Chat Application Setups

```
vagrant@Derp3:~/examples/sockets$ python client.py localhost 5000
Connected to remote host. Start sending messages
<You> UserKey: A WORD IS A WORD
Encrypted String:
5b 31 32 37 2e 30 2e 30 2e 31 3a 34 36 36 37 31 5d 20 65 6e 74 65 72 65 64 20 72 6f 6f 6d 0a
Decrypted String:
12 7f a9 8d 40 c7 b5 2b bb fc 1f 20 f6 3f a3 8d received [127.0.0.1:46671] entered room
:None
<You> hello overthere, how are you?
UserKey: A WORD IS A WORD
Original String:  0x68656c6c 0x6f206f76 0x65727468 0x6572652c
Encrypted String:  0xcbb5609c 0x51347bb8 0xe22375d 0x753d7d sending hello overthere, how are you?
:`@4{7]u=}
<You>
```

FIGURE D.1: RC6 Chat Application Client 1

```
Last login: Mon Jun 13 03:07:58 2016 from 10.0.2.2
vagrant@Derp3:~$ cd examples/sockets/
vagrant@Derp3:~/examples/sockets$ python client.py localhost 5000
Connected to remote host. Start sending messages
<You> UserKey: A WORD IS A WORD
Encrypted String:
0d 3c 28 27 31 32 37 2e 30 2e 30 2e 31 27 2c 20 34 36 36 37 30 29 3e 20 cb b5 60 9c 51 34 7b b8 0e 22 37 5d 00 75 3d 7d
Decrypted String:
<('127.0.0.1', 46670)> `@4{7]u=}:None 0f c9 received
<You>
```

FIGURE D.2: RC6 Chat Application Client 2

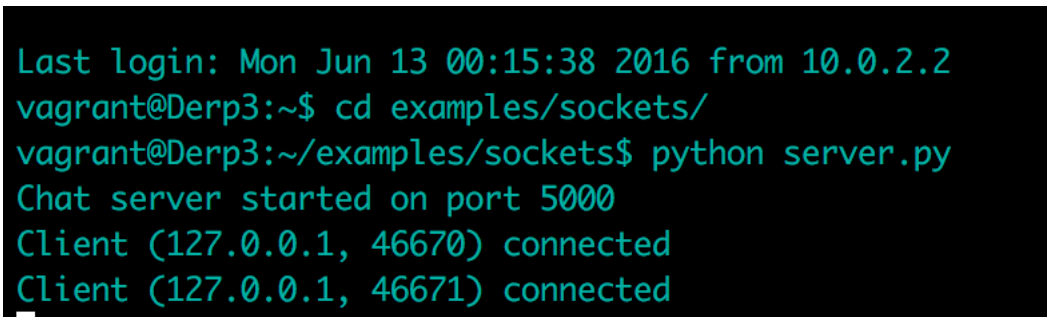

FIGURE D.3: RC6 Chat Application Server

Appendix E

Case Study: RC6 Chat Application Outputs

```
Confidence: 1.0
_____________________________________________
potential inputs
0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20 0x41 0x20 0x57 0x4f 0x52 0x44
     0x44
A WORD IS A WORDD
0x49 0x20 0x61 0x6d 0x20 0x76 0x65 0x72 0x79 0x20 0x73 0x6c 0x65 0x65 0x70 0x79
I am very sleepy
potential outputs
0xc1 0x60 0xcf 0x46 0xec 0x03 0xb9 0x01 0x29 0x1a 0x76 0x87 0xd0 0x29 0x35 0x2
_____________________________________________
Confidence: 0.75
_____________________________________________
potential inputs
0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20
A WORD IS
0x73 0x68 0x63 0x68 0x6e 0x61 0x6c 0x6a 0x53 0x68 0x68 0x41 0x73 0x21 0x52 0x44
shchnaljShhAs!RD
potential outputs
0xe3 0x01 0x29 0x1a 0xdf 0x43 0x46 0xec 0x27 0xd6 0x26 0x73 0x6c 0x65 0x96 0x49
_____________________________________________
...
```

LISTING E.1: Output for Message Two

```
Confidence: 1.0
_____________________________________________
potential inputs
0x41 0x20 0x57 0x4f 0x52 0x44 0x20 0x49 0x53 0x20 0x41 0x20 0x57 0x4f 0x52 0x44
A WORD IS A WORD
0x50 0x6c 0x73 0x20 0x72 0x65 0x73 0x70 0x6f 0x6e 0x64 0x2c 0x20 0x77 0x68 0x20
     0x79 0x20 0x79 0x6f 0x75 0x20 0x6e 0x6f 0x20 0x72 0x65 0x73 0x70 0x6f 0x6e
     0x20 0x64 0x20 0x74 0x6f 0x20 0x6d 0x65 0x3f 0x21 0x20 0x20 0x20 0x20 0x20
     0x20 0x20
Pls respond, wh y you no respon d to me?!
potential outputs
0x33 0x63 0x85 0xb2 0x1a 0x0f 0xd1 0x7f 0x2f 0x8b 0x7a 0x86 0x6b 0xed 0x67 0x35
     0xe6 0x9c 0x99 0x69 0x56 0x23 0xa7 0xf5 0x93 0xe2 0xaf 0x12 0xb6 0x3f 0x0f
     0xed 0x2e 0x73 0xf5 0x9a 0x 0 0xx8 0x0b 0xb0 0xb0 0x6c 0xcb 0xfa 0xaf 0xdf
     0x65 0x42 0xb2 0x04
_____________________________________________
```

LISTING E.2: Output for Message Three

Bibliography

[1] Jeff Atwood. *The Infinite Space Between Words*. Visited on 26/05/2016. 2014. URL: https://blog.codinghorror.com/the-infinite-space-between-words/.

[2] Davide Balzarotti et al. "Efficient Detection of Split Personalities in Malware." In: Visited on 02/06/2016.

[3] Sion Berkowits. *Pin - A Dynamic Binary Instrumentation Tool*. Visited on 27/05/2016. 2012. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[4] Carl Friedrich Bolz et al. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. Visited on 26/05/2016. ACM. 2009, pp. 18–25.

[5] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. "Towards automated protocol reverse engineering using semantic information". In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. Visited on 22/05/2016. ACM. 2014, pp. 51–62.

[6] Juan Caballero et al. "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering". In: *Proceedings of the 16th ACM conference on Computer and communications security*. Visited on 16/05/2016. ACM. 2009, pp. 621–634.

[7] Joan Calvet, José M Fernandez, and Jean-Yves Marion. "Aligot: cryptographic function identification in obfuscated binary programs". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. Visited on 13/05/2016. ACM. 2012, pp. 169–182.

[8] J Daemen and V Rijmen. "AES Proposal Rijndael, the First Advanced Encryption Standard". In: *Candidate Conference*. Visited on 15/05/2016. 1998.

[9] NIST FIPS. "46-3. Data Encryption Standard". In: *Federal Information Processing Standards, National Bureau of Standards, US Department of Commerce* (1977). Visited on 15/05/2016.

[10] NIST FIPS. "Digital Signature Standard (DSS)". In: (2000). Visited on 16/05/2016.

[11] Robert M Gray. *Entropy and information*. Visited on 25/05/2016. Springer, 1990.

[12] Felix Gröbert, Carsten Willems, and Thorsten Holz. "Automated Identification of Cryptographic Primitives in Binary Programs." In: *RAID*. Vol. 6961. Visited on 16/05/2016. Springer. 2011, pp. 41–60.

[13] Carl Hadley. *1.5 Million Home Automation Systems Installed in the US This Year*. Visited on 11/05/2016. 2012. URL: https://www.abiresearch.com/press/15-million-home-automation-systems-installed-in-th/.

[14] Ralph Langner. "Stuxnet: Dissecting a cyberwarfare weapon". In: *Security & Privacy, IEEE* 9.3 (2011). Visited on 16/05/2016, pp. 49–51.

[15] Xin Li, Xinyuan Wang, and Wentao Chang. "CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution". In: *Dependable and Secure Computing, IEEE Transactions on* 11.2 (2014). Visited on 16/05/2016, pp. 101–114.

[16] Zhiqiang Lin et al. "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution." In: *NDSS*. Vol. 8. Visited on 20/05/2016. 2008, pp. 1–15.

[17] Christian Ludloff et al. *X86 Opcode and Instruction Reference*. Visited on 23/05/2016. URL: http://ref.x86asm.net/#column_mnemonic.

[18] Noé Lutz. "Towards revealing attacker's intent by automatically decrypting network traffic". In: *Mémoire de maitrise, ETH Zürich, Switzerland* (2008). Visited on 16/05/2016.

[19] Mark S Mayzner and Margaret Elizabeth Tresselt. "Tables of single-letter and digram frequency counts for various word-length and letter-position combinations." In: *Psychonomic Monograph Supplements* (1965). Visited on 24/05/2016.

[20] odzhan. *Tinycrypt*. Visited on 06/06/2016. URL: https://github.com/odzhan/tinycrypt/.

[21] PB Pathak and Yeshwant Nanded. "A Dangerous Trend of Cybercrime: Ransomware Growing Challenge". In: (2012). Visited on 11/05/2016.

[22] RL Rivest et al. *The RC6 block cipher. v1. 1, August 20, 1998*. Visited on 11/06/2016.

[23] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. *Cryptographic communications system and method*. US Patent 4,405,829. 1983.

[24] Jonathan Salwan and Florent Saudel. "Triton: Framework d'exécution concolique et d'analyses en runtime". In: (). Visited on 02/06/2016.

[25] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *Security and privacy (SP), 2010 IEEE symposium on*. Visited on 15/05/2016. IEEE. 2010, pp. 317–331.

[26] Adi Shamir and Nicko Van Someren. "Playing 'hide and seek'with stored keys". In: *Financial cryptography*. Visited on 01/69/2017. Springer. 1999, pp. 118–124.

[27] Robert Endre Tarjan. "Fast algorithms for solving path problems". In: *Journal of the ACM (JACM)* 28.3 (1981). Visited on 21/05/2016, pp. 594–614.

[28] International Telecommunication. "ITU ICT Facts and Figures – The world in 2015". In: *Telecommunication Development Bureau* 1 (2015). Visited on 11/05/2016, pp. 1–2. URL: http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx.

[29] Duygu Sinanc Terzi, Ramazan Terzi, and Seref Sagiroglu. "A survey on security and privacy issues in big data". In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. Visited on 09/06/2016. IEEE. 2015, pp. 202–207.

[30] Jordi Tubella and Antonio Gonzalez. "Control speculation in multithreaded processors through dynamic loop detection". In: *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*. Visited on 21/05/2016. IEEE. 1998, pp. 14–23.

[31] Zhi Wang et al. "ReFormat: Automatic reverse engineering of encrypted messages". In: *Computer Security–ESORICS 2009*. Visited on 16/05/2016. Springer, 2009, pp. 200–215.

[32] AF Webster and Stafford E Tavares. "On the design of S-boxes". In: *Advances in Cryptology—CRYPTO'85 Proceedings*. Visited on 16/05/2016. Springer. 1986, pp. 523–534.