Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Noé Lutz

Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic

A joint project between the ETH Zurich and Google, Inc.

Master Thesis MA-2008-08
January 2008 to July 2008

ETH Tutor: Bernhard Tellenbach
Google Tutors: Niels Provos and Thomas Dübendorfer
Supervisor: Prof. Bernhard Plattner

Abstract

Researchers and commercial security companies are constantly improving their techniques to detect and prevent malicious software (malware) proliferation. Unfortunately, at the same time malware authors continuously improve their techniques to evade detection. One recent development in this arms race is the use of encrypted network communication by malware authors to prevent the analysis of malware capabilities and hide malicious activities.

To date, researchers have manually analyzed malware binaries to reverse engineer their decryption algorithms. Manual malware analysis, however, is very labor intensive and does not scale as the number of malware binaries that use encryption increases. To address these shortcomings of manual analysis we present a *generic* and *automatic* tool for the decryption of encrypted communication received by a binary. Our tool does not break cryptography but leverages the fact that the binary decrypts the encrypted input it receives during its execution. We use a dynamic binary analysis approach to run the binary and identify *where* and *when* the decrypted input is located in the system's memory.

We provide evidence that our tool effectively locates decrypted input for various linux cryptographic libraries. We also present a case study of the ability of our tool to decrypt the network communication of a real malware bot sample.

We argue that our approach can be used as a tool for revealing the intent of attackers that try to masquerade their activities using encrypted communication.

Contents

1 Introduction

1.1 Motivation

Researchers and commercial security companies constantly improve their techniques to detect and prevent malicious software (malware) proliferation, while malware authors routinely improve their techniques to evade detection. An example of this arms race is the use of binary packing and encryption to prevent straight forward, signature-based detection. More recently, malware has been observed that encrypts network communication rendering signature-based network intrusion detection more challenging than in the past. In the case of botnets, encrypted traffic prevents automatic analysis of the bot's capabilities by observing network payloads. While in some cases the malware authors use simple encryption schemes, such as a substitution cipher, in others they use well-known cryptographic encryption algorithms such as the RC4 algorithm [3].

The Storm and Kraken bots are the most recent and widely publicized examples of malware that encrypt their communication. In order to study the behavior and capabilities of these bots and to effectively slow their spreading, it is important to be able to decrypt their network traffic. In their analysis of the rustock rootkit and spam bot, Chiang and Lloyd [3] demonstrate the difficulty of manual malware analysis. The use of manual malware analysis to decrypt network traffic does not scale with the increasing number of malware binaries that use custom encryption techniques. In addition to the rapidly growing number of malware in the wild, analysis is also time-sensitive. In some cases malware authors very quickly adapt their encryption scheme after it has been broken. As an example, the decryption algorithm used by Kraken version 316 was published on a blog in April of 2008 [11][19]. Just ten days later another security blogger reported a new version of the Kraken bot (v.317) [25]. This version uses a different encryption scheme, rendering the manual analysis of previous Kraken versions irrelevant. While we have no evidence to suggest that the motive for updating the bot was publication of its decryption algorithm, a cause-effect relationship seems likely here.

To effectively cope with an increasing number of malware binaries that use encryption and with the quick response time of malware authors an *automatic* binary analysis approach appears necessary.

1.2 Problem Statement

This thesis seeks to alleviate the bottleneck incurred by manual analysis of binaries that use encrypted network communication by automating the analysis process. The goal is to automatically decrypt any encrypted communication received by the binary under analysis.

1.3 Solution Approach

Rather than attempting to break cryptography we leverage the fact that the binary decrypts the encrypted input it receives during its execution. In our research, the challenge is to know *when* the input gets decrypted and *where* the decrypted data is located in the memory. Apart from technical challenges, the difficulty of this problem is to find a generic solution that is not tailored to a particular malware binary or cryptographic algorithm. Instead we strive for a solution that works for a multitude of different encryption algorithms and implementations.

The main contribution of our research is the design and implementation of an automatic binary analysis tool that can effectively decrypt program input for a variety of cryptographic algorithms and implementations. We use dynamic binary instrumentation to monitor a binary's execution as it decrypts the encrypted input. The first step of our analysis extracts various features from the binary's execution, such as: information regarding memory access patterns and the control flow including program loops, and function calls. In the second step, we search the extracted data for features indicative of the decryption process. Once these features are extracted, we search for information entropy decreasing loops that use a proportionally high number of integer arithmetic operations. This heuristic is based on three observations: First, decryption most likely happens in a loop. For example, in the case of a block cipher it is likely that there is a loop over the input buffer that decrypts the input block-by-block. Second, encrypted data is likely to have a higher information entropy than decrypted data. This observation certainly holds true for cryptographically secure algorithms. Third, most cryptographic algorithms are related to number theory and so naturally involve integer arithmetic operations. Once we pinpoint the potentially multiple locations of the decryption process, we output all memory locations that we presume contain the decrypted data.

We use dynamic tainting techniques to track the dependencies of encrypted input in memory. Taint-tracking helps us to find candidate memory locations that contain the decrypted data, effectively reducing the search space.

1.4 Evaluation and Results

To explore the effectiveness of our approach we developed an implementation of our design using Valgrind, a dynamic binary instrumentation framework [17]. Our experiments demonstrate that our implementation successfully decrypts the encrypted input of common Linux programs that use various cryptographic algorithms and implementations. For example, we can decrypt content fetched by curl over an HTTPS connection.

In order to show that our approach also applies to malicious software

we ran our analysis tool on a version of the Kraken bot. The Kraken bot uses a custom encryption algorithm described by Ligh [11]. The experiment shows that our analysis tool successfully decrypts Kraken's encrypted network traffic.

1.5 Scope and Limitation

By using binary instrumentation instead of whole system instrumentation we limit our analysis to binaries that decrypt their input in user mode and do not instrument code that is running in the kernel. A limitation of Valgrind is that it only runs on Linux and not on Windows. This limits our evaluation of malicious binaries that run on Linux or in Wine. We leave it for future work to port our proof-of-concept implementation to Windows.

The performance of our approach depends on input taint-tracking since it greatly reduces the search space of decrypted input. Cavallaro et al. [12] illustrated with practical examples how dynamic taint analysis can be evaded. We did not investigate how these evasion techniques can be detected or circumvented.

Making the assumption that the decryption algorithm decreases memory entropy introduces another limitation. Simple substitution ciphers, such as the Caesar cipher, typically do not affect entropy, i.e. encrypted and decrypted messages have the same entropy. Since substitution ciphers can easily be broken we believe that malware authors will prefer more secure, readily available encryption tools that do change entropy. If needed, our analysis tool could be extended to support multiple heuristics that look for different features. For example, the Caesar decryption loop could be detected by doing a frequency analysis on the data before and after each loop.

1.6 Structure of this Report

This report is organized as follows: The next section presents related work. Section 3 discusses the design and architecture of our approach. Section 4 talks about our implementation. Section 5 presents the experimental setup and evaluation results, and Section 7 concludes our report.

2 Related Work

In this section, we discuss related work that gives background on the general problem of malware analysis and the specific problem addressed in this thesis (the decryption of network communication).

Generally, malware analysis encompasses two main techniques: static and dynamic analysis. Static analysis uses reverse engineering techniques to disassemble the malware and extract its features. Early work in static

malware analysis has tended to focus on approaches to malware detection [5]. Unfortunately, static analysis of malicious software has been challenged by obfuscation techniques used by malware authors for decades [26]. Christodorescu et al. analyze the semantic behavior of code as a way to thwart some binary obfuscation methods [4]. More recently Moser et al. provide evidence of the limitations of static analysis by showing that even advanced semantics-based malware detectors can be evaded [14]. Their work demonstrates that static analysis techniques alone are no longer sufficient for malware identification. The dynamic analysis approach circumvents the problem of binary obfuscation by running the malware binary and extracting information from its execution rather than from its code. Recent research efforts around automatic malware analysis have therefore been biased towards dynamic analysis. Dynamic analysis comes with its own limitations most notably performance issues induced by instrumentation and detection of the analysis by the malicious binary. Kirda et al. [9] introduce a novel technique to the detection of spyware based on the characterization of spywarelike behavior. They present a hybrid approach combining both static and dynamic analysis to evaluate the malicious behavior of spyware. CWSandbox [28] analyzes the execution of a malware binary by observing the sequence of invoked system calls. Portokalidis et al. present Argos, a more fine grained dynamic analysis environment designed to automatically detect zero-day attacks [21].

In this thesis we opted for a dynamic analysis approach. Unfortunately while a large body of research exists on dynamic analysis techniques, the problem of decrypting network communication has received considerably less attention. To the best of our knowledge there is no published work that takes an automatic and generic approach to decrypting network traffic.

Chiang and Lloyd present a detailed, manual analysis of the Rustock Rootkit and Spam Bot [3]. Their static analysis of the malware focuses on the necessary steps to decrypt the communication over the command and control channel between the spam bot and its bot master. Our approach achieves the same decryption without the limitations of manual analysis to enable scaling.

Given a particular cryptographic algorithm, there are ways to automatically find cryptographic keys stored on a machine's hard drive [24] or in a program's memory [7]. However, in the case of malware, the cryptographic algorithm is in general unknown. Our approach tries to be more generic and not assume the use of any particular cryptographic algorithm.

3 Design and Architecture

In this section we describe the design and architecture of our approach. We first elaborate the assumptions that form the basis of our approach in

Section 3.1. An overview of our system is presented in Section 3.2. Sections 3.3, 3.4 and 3.5 provide a detailed discussion of the specific techniques used in our approach. Finally, Section 3.6 puts it all together.

3.1 Approach and Assumptions

Given an unknown program to analyze, we wish to automatically decrypt all encrypted input it receives from either the network or the file system. Some malware authors use simple substitution ciphers that can be broken automatically. However, focusing on breaking encryption algorithms used by malware authors presently would not be very effective in the future, as malware authors would quickly adapt their techniques and use well known, cryptographically secure algorithms. In this research we do not try to break cryptography. Instead we aim for a solution that works for any kind of encryption algorithm including cryptographically secure algorithms. To achieve this objective, we make a series of assumptions. One fundamental assumption of this research is that the program decrypts the encrypted input it receives.

While we are not aware of any malware that violates this assumption it may not hold true. Imagine malicious software that receives encrypted input but instead of decrypting the input, the program is written in such a way that it acts upon encrypted messages. In this scenario bot commands could be encrypted strings instead of cleartext strings. Due to the theoretical existence of this type of malware we did not consider this case in our design.

Given this assumption, our approach relies on the program under analysis to decrypt the encrypted input for us. To determine where the decryption process occurs within the program we utilize a program analysis approach. Automatically analyzing malicious programs is a challenging task. There is no source code available and thus the analysis has to be done on the malicious binary, which are often obfuscated to purposefully confuse examination.

There are two main approaches to automatic program analysis: static analysis and dynamic analysis. As previously stated, malicious software binaries are often packed or obfuscated to evade purely static analysis and reverse engineering of the source code. Dynamic analysis circumvents this problem because the instructions have to be unpacked before they can be executed. This advantage is the primary reason for which we decided to use dynamic analysis in this research.

Dynamic analysis involves instrumenting the binary with analysis code. The analysis code runs as part of the program and collects information about its execution. The analysis code should not affect the program's normal execution other than possibly slowing it down. By slowing down the malware's execution, we risk that it detects the ongoing analysis and stops exhibiting the interesting behavior - in our case the decryption of encrypted input. We assume that the program under analysis does not detect the analysis pro-

cess and continues to exhibit the behavior of interest. Our current design does not take any particular steps to avoid such detection. Ferrie discusses several methods for detecting virtual machines and emulators that are frequently used for dynamic analysis [6]. We leave it for future work to make our analysis approach more stealthy.

In addition, we assume that we can repeat the analysis for a particular binary. This supposes either that the other communication endpoint, e.g. the bot master, is still up and running or that we can replay previously recorded traffic. Running malicious software and letting it communicate with a live bot master may negatively impact Internet users, e.g. malware could start sending spam. Therefore, we prefer replaying existing network traffic when possible. If the software to analyze uses randomized encryption, i.e. encryption that uses nonces or initialization vectors, replaying network traffic is not feasible. In this case we limit the malware's network accessibility and only allow the encrypted network traffic that we want to decrypt.

3.2 Design Overview

Based on the assumptions described above, our approach is to dynamically analyze the binary to extract the decrypted input from its memory. Given this approach, the problem we are trying to solve can be reformulated as: *when* does the encrypted input get decrypted and *where* is the decrypted input located in memory?

At a high level, the system that we designed to address this problem is a three step process. The first step runs the binary and extracts key features from its execution. In the second step, an offline detection algorithm uses the extracted features to find where the decryption process occurs and where the decrypted input is located in the memory. Finally, in the third step, the binary is dynamically analyzed a second time, but this time with knowledge of the time and place at which decryption occurs. The output of the third step is the decrypted version of the input. Figure 1 shows an overview of this system.

We chose this simple and flexible architecture because it allows us, in an exploratory phase of the research, to investigate multiple features and detection algorithms. This framework can be easily extended by adding new features and/or modifying the detection algorithm to iteratively find an effective algorithm. In addition this approach simplifies the design of the detection algorithm. Dynamic analysis and feature extraction operate at the instruction level. By separating the feature extraction from the detection algorithm, we avoid needing to write the detection algorithm at the instruction level.

One disadvantage of this approach is that the dynamic analysis extracts features from the entire program and not just from the decryption process. This fact negatively impacts performance because more information is pro-

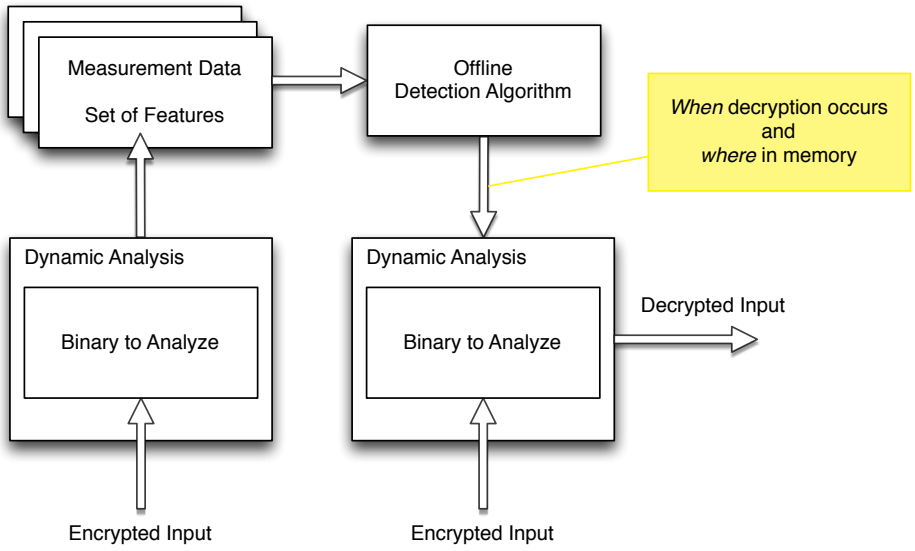

Figure 1: System Overview

cessed than is really needed. In addition, by running the binary twice, we prolong the time required for analysis. To cope with this decrease in performance, once an effective detection algorithm has been designed, one could merge the three steps of our system into one single step.

The following sections describe the evolution of the features extracted and the detection algorithm as they were modified and improved. Before this detailed discussion, we briefly describe a technique used to reduce the number of candidate memory locations that may contain the decrypted input.

3.3 Memory Tainting

To help identify where in the program's memory the decrypted input is located, we use memory tainting techniques. Memory tainting helps us to address the *where* question by reducing the search space and limiting the number of candidate memory locations that may contain the decrypted input. This technique also helps to focus further analysis on the parts of the program that touch (i.e. read or write) tainted input data, which addresses the *when* question. For example, parts of the program that do not read tainted data are unlikely to play a major role in the decryption process.

All encrypted input is marked as tainted upon reception. Taint tracking is used to taint every memory location that depends on already tainted data. Assuming that the decrypted input depends on the encrypted input, taint

tracking will result in the decrypted input being marked as tainted as well. This result is illustrated as a simple example in Figure 2.

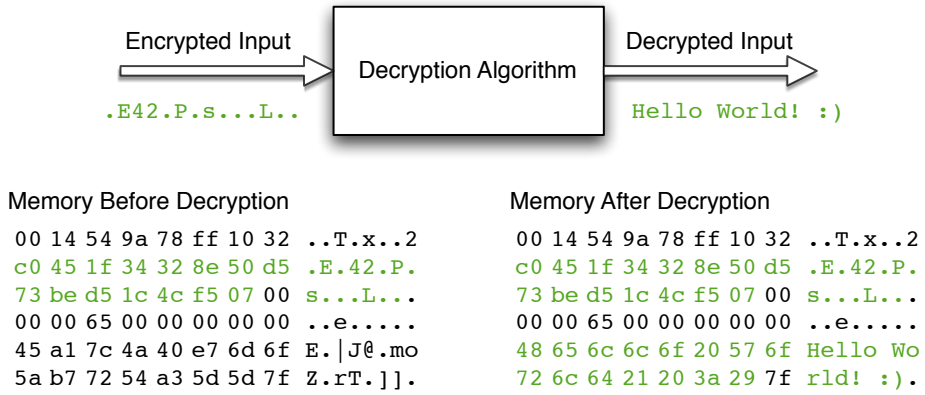

Figure 2: Example of Memory Tainting.

Our taint tracking approach is similar to TaintCheck [18]. Here we describe the aspects of taint tracking that are important to our analysis.

3.3.1 Taint Source

In the context of this research the taint source corresponds to the encrypted input that the program reads. Encrypted input is read from either the network or the file system. It is possible to selectively taint data read from these two sources to avoid tainting of non-encrypted input. In this research tainting reduces the search space. Over-tainting would unnecessarily enlarge the search space and negatively impact performance. At the time of this writing filtering rules are defined manually. For example, network traffic can be filtered by indicating an IP address and/or port number whitelist. In future work we will focus on automatically selecting encrypted input for tainting.

3.3.2 Taint Propagation

After encrypted data is tainted upon reception, we propagate tainting to every memory location that depends on already tainted data. To do this we track all instructions of the program that manipulate data and determine if the result must be tainted. Special cases in which additional data must be tainted or the result untainted were previously described and implemented in this work [18].

3.3.3 Tainted Memory

We store the taint status of each memory byte and register in the so called shadow memory. The shadow memory is a data structure that holds a four byte value for each memory address. The four bytes are used to hold the taint status and to indicate the origin, i.e taint source, of the tainted data.

3.3.4 Limitations of Memory Tainting

Memory tainting can be evaded. Cavallaro et al. present simple techniques to effectively evade memory tainting [12]. In our research the use of these techniques would lead to the decrypted input being untainted. Should tainting be evaded all memory locations must be considered as potential locations for the decrypted input which would negatively impact performance. At this point our research does not attempt to detect taint evasion. To the best of our knowledge, no malware uses taint evasion to date.

3.4 Feature Extraction

The challenge of feature extraction is to identify the features (e.g. control flow patterns, data access patterns, etc.) necessary for the detection of the decryption process. We need to find features that are invariant across most decryption algorithms including unknown decryption algorithms used by malware authors.

We used binary instrumentation as a tool to extract features from the program's execution. Binary instrumentation operates at the instruction level and is inconvenient for finding features indicative of decryption. We use binary instrumentation to extract higher level features from low level instructions.

3.4.1 Function Detection

Each function in a program typically performs a very specific task. Therefore, functions appear to be at a good level of abstraction to search for tainted data access and control flow patterns. We dynamically detect function calls and keep track of the program's call stack. Function calls are detected by instrumenting jump, call and return instructions and monitoring the stack and frame pointers. In the case of static or inline functions, the compiler may not follow calling conventions. In these cases our detection may not work, but we do not see this as problematic, since it simply results in misclassification of the body of the inline or static function as part of the calling function. We have found that our simple detection method works well in most cases.

As a first step towards detecting where in the program decryption occurs we keep track of the functions that read or write tainted memory locations.

We call functions that touch tainted memory locations (i.e. memory that depends on the encrypted input) *candidate functions*. Functions that do not touch tainted data are probably not part of the decryption process and can therefore be ignored. Table 1 shows the fraction of candidate functions for common Linux cryptographic libraries. These numbers were obtained by running these tools in our analysis environment and allowing them to decrypt previously generated encrypted input.

For symmetric encryption like AES and Blowfish we find less than ten candidate functions. In contrast, OpenSSL and GnuPG contain an increased number of candidate functions, which suggests that more operations are performed on tainted data. This result is expected since both OpenSSL and GnuPG use an additional step to decrypt encrypted data. The first step in their decryption process employs asymmetric cryptography to generate keys for the second step. In the second step symmetric cryptography is applied to decrypt the encrypted input.

Looking only at candidate functions significantly reduces the number of locations in the program that are potentially part of the decryption. Nevertheless, this approach does not suffice to automatically identify exactly when during the program's execution decryption occurs.

Table 1: Fraction of *candidate functions*, i.e. functions that operate on tainted memory.

Decryption Tool	Number of Functions		Fraction of
	Total	*Candidate*	*Candidate Functions*
libgcrypt (AES)	791	3	0.38%
bcrypt (Blowfish)	197	5	2.54%
GnuPG	609	45	7.39%
cURL (OpenSSL)	1314	39	2.97%

3.4.2 Control Flow Graph

In order to pinpoint more accurately when decryption occurs during the program's execution we investigate the program's control flow graph (CFG). Each vertex in the CFG represents a basic block, i.e. consecutive code instructions without any jumps or jump targets. Edges in the control flow graph are directed and represent jumps in the control flow. Figure 3 shows the correspondence between a simple C code and its CFG. In particular we analyze the CFG of candidate functions. The idea is to find sub-graphs within the control flow graph that are indicative of the decryption process or even of a particular algorithm. Once these sub-graphs are found, we can create a list of known sub-graph patterns to search for within the program's

CFG. Figure 4 shows an example of two CFGs that share a common sub-graph. We choose to compare patterns at the level of the CFG rather than the instruction level because two implementations of the same algorithm are much more likely to share common sub-graphs than to share common code.

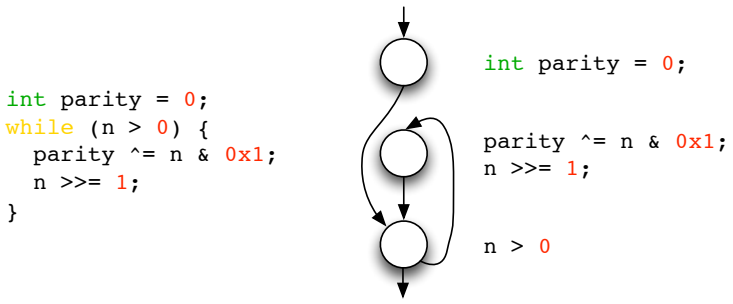

Figure 3: Shows correspondence between code and its control flow graph.

Similar ideas were used by Kruegel and Bruschi to detect polymorphic worms and self-mutating malware [10], [2]. In their work, Bruschi et al., construct a labelled control flow graph of a malware binary. They then search for common sub-graphs between this CFG and the CFG of known malware, thus reducing the problem of malware detection to the sub-graph isomorphism problem. The sub-graph isomorphism problem is a well known NP-complete problem but can, in the majority of cases encountered in this context, be resolved efficiently [2].

We cannot rely on static analysis to generate the control flow graph since the binary may be packed or obfuscated. Packing techniques may use self-modifying code which prevents the CFG from being built using static analysis. Instead, we dynamically reconstruct the CFG during the program's execution. Our dynamic CFG differs from a traditional CFG in that it only contains basic blocks (BB) that are executed. Since we assume that the decryption process *is* executed, it must be present in our dynamic control flow graph.

We mention above that we choose to compare patterns at the level of the CFG since the underlying instructions for two implementations of a particular algorithm often vary. Conversely, two code sections may have the same sub-graphs but contain very different instructions. Therefore, comparing patterns based solely on the dynamic CFG is too coarse grained of an approach. To resolve this issue, we label each vertex in the CFG according to the properties of the instructions of the corresponding basic block. We extend the labeling method presented by Kruegel et al. in [10]. Instructions are categorized into instruction classes based on their behavior. For example, all instructions that perform comparison are grouped into the instruction class "Comparison". Table 2 lists the instruction classes that we

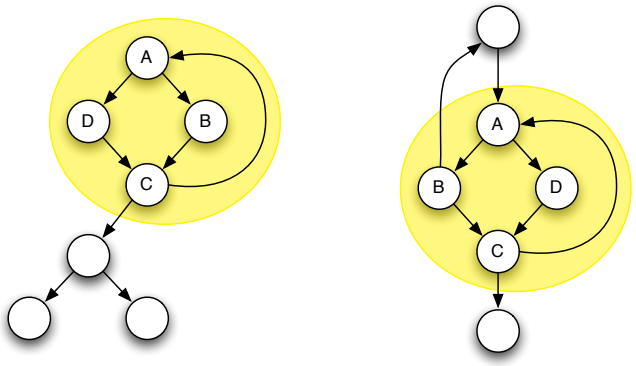

Figure 4: Example of two control flow graphs sharing a common subgraph.

distinguish. We created a separate class for the xor instruction because it is frequently used in cryptographic algorithms. The integer arithmetic and xor classes are the most useful ones to identify cryptographic algorithms. We label each basic block with the set of instruction classes that are executed within it. Importantly, only instructions that touch tainted data are considered in the labeling process, since these are the instructions that modify the encrypted data and determine the basic block's interesting behavior. In addition, we label each vertex with the number of tainted bytes read and written by the corresponding basic block. Figure 5 shows an example of an annotated control flow graph.

Table 2: List of instruction classes that categorize instructions.

Instruction Classes
Integer Arithmetic
Xor
Floating Arithmetic
Logic
Comparison
Conversion [a]

[a] Our instrumentation framework is type safe and therefore provides type conversion operators.

The tool described above extracts an annotated CFG of a binary's execution. The goal is to identify template sub-graphs that represent decryption and can be searched for within the binary under analysis. In order to define these templates, we used our tool to analyze multiple common Linux binaries that use encryption. By manually scrutinizing the CFGs extracted from these binaries, we noted, not surprisingly, that loops are a recurring

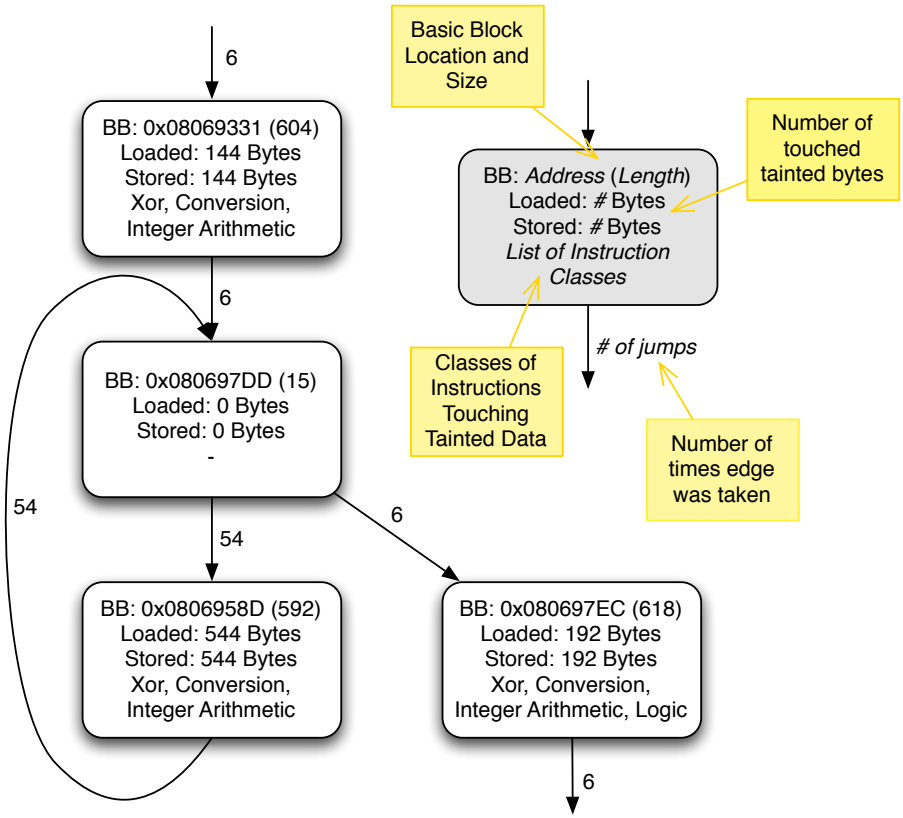

Figure 5: Example of an annotated control flow graph, extracted from the libgcrypt Rijndael (AES) implementation.

feature in decryption algorithms. We therefore proceeded to augment our labeling of CFG vertices with information regarding loops.

3.4.3 Loop Detection

Most encryption and decryption algorithms use loops such as `for` or `while` loops in their implementation. For example, block cipher implementations often use multiple rounds of the same basic operations to decrypt a block. Loops are also used for the decryption of variable length messages. Messages longer than block size are split into equally sized blocks and iteratively decrypted.

We use a well known algorithm to dynamically detect loops during the program's execution [27]. Loops are found by detecting backwards edges within the control flow graph. A loop, identified by address T, is present within a program when one or more backward branches point to address T. If multiple branches point to the same address T, we consider all of these branches to close the same loop. The body of the loop encompasses all instructions between address T and the highest address with a backward branch to T denoted as B. Figure 6 shows how T and B are defined.

To handle nested loops and keep track of loop statistics, we use a current loop stack. This stack contains all loops being currently executed. The top of the stack corresponds to the innermost loop. Calls to subroutines within a loop are considered to be part of that loop, so that the body of the function called becomes part of the body of the loop. This property also holds for recursive functions, i.e. loops do not stop at function boundaries.

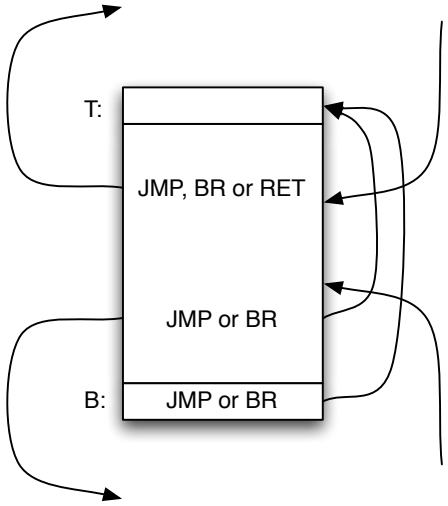

Figure 6: Static view of a loop, reproduced from the original paper [27].

Loops used in decryption algorithms have specific features that further

help to identify the decryption process. As an example, the loop that iterates over block cipher rounds operates on a small number of tainted bytes, roughly equal to the block size, and iterates for a constant number of rounds. In addition, the loop that iteratively calls the block cipher on sequential blocks usually uses a well known block cipher mode of operation to combine adjacent block content. By identifying loops with these features characteristic of decryption, we can further narrow the search space for finding the decryption process.

To detect loops that iterate for a constant number of rounds we simply keep track of the minimum and maximum number of iterations for each loop's execution. If the min and max number of iterations are the same, then the number of iterations for that loop is constant. In order to detect additional features, we define the following loop types.

Accumulator loops write tainted data during their first iteration and then modify the same memory location during subsequent iterations. A typical example of an accumulator loop is a loop that computes the sum of an integer array.

Out Swipe loops write roughly the same amount of tainted data to *unique locations* with each iteration, except for the first or the last iteration that may taint more, e.g. temporary variables. An example of a *Out Swipe* loop is the loop that iterates over the basic blocks and encrypts each basic block individually.

In and Out Swipe loops are similar to *Out Swipe* loops but read *and* write the same amount of tainted data during each iteration, as opposed to simply writing tainted data.

Memory Copy loops are loops that only copy tainted data around. It turns out that memory copy loops are very frequent in code. We distinguish these loops from other loops to avoid misclassifying memory copy loops as decryption loops.

Block ciphers frequently use a power of two as their block size. We further categorize any *In and Out Swipe* or *Out Swipe* loops for which the number of unique memory locations touched per iteration amounts to a power of two as $Swipe^2$.

Given these loop labels, the loop that iterates over block cipher rounds may be labeled as a *Accumulator* loop that uses a constant number of iterations. Similarly, the loop that iteratively calls the block cipher on sequential blocks may be labeled as a *Out Swipe* loop and even a *In Out Swipe* or $Swipe^2$ type loop.

In order to detect these loop types, we count the number of unique tainted memory locations read and written for each iteration. This information is aggregated at the loop execution level, where we keep track of only

the minimum and maximum number of tainted bytes touched across all iterations. This min and max information allows us to assign a loop type to each loop following its execution. Refer to Appendix C for a detailed description of the detection algorithms for the Swipe and Accumulator type loops based on this min and max principle. We postpone discussion of the memory copy algorithm until later in this report when we introduce analyzing loop input and output.

We now add the results of our loop detection and loop feature extraction to our annotated control flow graph. Figure 7 shows the previous control flow example updated to include this additional information (Figure 5).

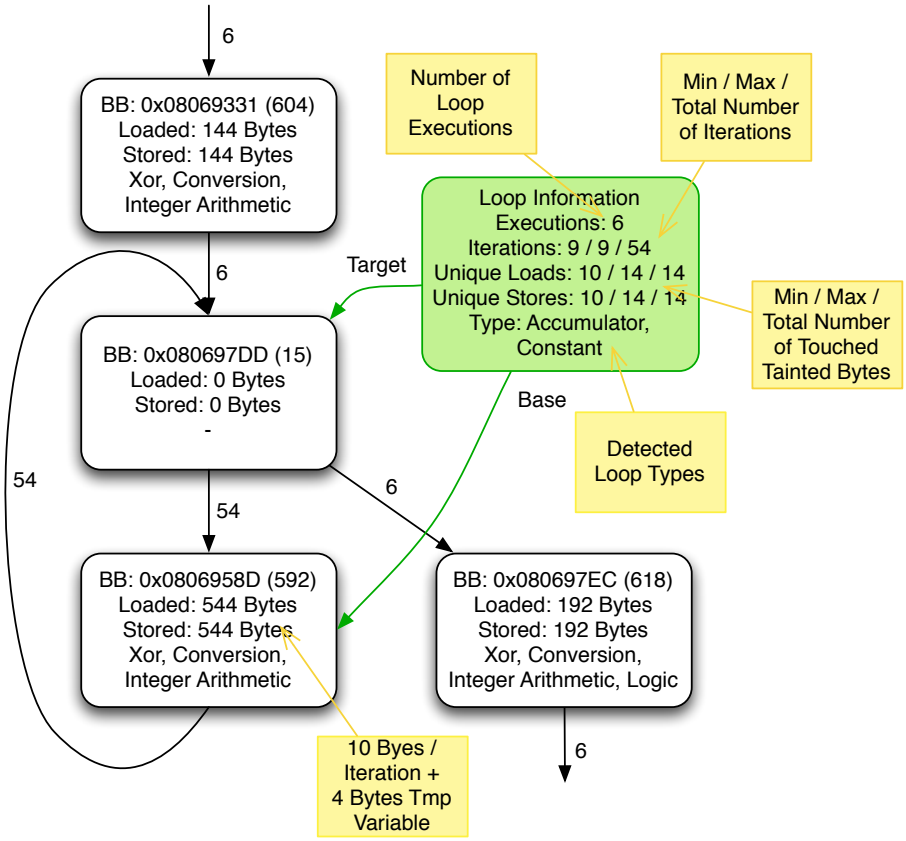

Figure 7: Update control flow graph with loop annotations.

3.4.4 Cryptographic Constants

Cryptographic algorithms frequently use lookup tables to implement substitution and permutation boxes (S-boxes and P-boxes). A typical AES im-

plementation, for example, uses four lookup tables containing 256 integers each. In the case of AES these lookup tables contain very specific constants. This fact can be used to identify whether or not the binary under analysis is using AES. Knowing which cryptographic algorithm is being used brings us one step closer to detecting when the input undergoes decryption.

To test the effectiveness of this method we considered constants used by: AES, Blowfish, Twofish and DES. A hash table is used to map cryptographic constants to their corresponding algorithm. Constants used by multiple algorithms are removed (see Table 3). We instrument every memory LOAD and lookup the loaded bytes in the hash table. For every match we increment a counter for the particular algorithm that matched. At the end of an analysis we have the number of matches for every algorithm.

With this simple approach we can effectively detect AES, Blowfish and Twofish implementations. To check for false positives we analyzed various Linux tools that do not use one of these algorithms. For example, we used `gzip` to compress a file that contains randomly generated data. No false positives were found for these three algorithms. This approach, however, does not work for the DES algorithm. The constants used by DES seem to occur frequently in programs unrelated to DES.

We decided to abandon this approach for two reasons. First, this approach assumes that we already know the algorithm and the cryptographic constants used by the malware author. Second, and more importantly, algorithms like Blowfish and Twofish can easily be modified to use other constants without affecting their security. This flexibility applies to every S-box and P-box implementation used in a Feistel network.

Table 3: Cryptographic Constants

Algorithm	Number of Constants	Unique Constants
AES	3072	3060
Blowfish	1042	1042
Twofish	1024	1020
DES	512	480

3.5 Heuristics and Detection Algorithm

So far we have described a tool that extracts specific features from a program's execution in order to narrow our search for where and when the decryption process occurs. Finding these features, however, is not sufficient to decisively locate the decryption process. In this section, we describe a heuristic that we use in combination with these extracted features to reliably identify the decryption process.

Our heuristic is based on three observations.

Loops As argued in the previous section, loops are an important component of cryptographic algorithms.

Entropy The decryption process typically decreases the information entropy of tainted memory. In the case of cryptographically secure algorithms, encrypted data typically has higher information entropy. This observation does not hold for simplistic algorithms, such as the Caesar cipher, that do not affect information entropy. Therefore, our heuristic will not work for algorithms like the Caesar cipher. Nevertheless, we believe that the approach of dynamically extracting features from a program's execution could still be used in combination with another heuristic to decrypt these simple cryptographic algorithms.

Integer Arithmetic Cryptographic algorithms heavily use integer arithmetic, in particular xor operations.

Based on these observations, we search for loops that decrease the information entropy of tainted memory and use integer arithmetic and xor operations. To track changes in tainted memory entropy we compare the information entropy of the tainted memory before and after the loop's execution.

3.5.1 Loop Input and Output

We define the *input* of a loop to be all *first* memory loads, i.e. the value of all memory locations read by the loop body but not (yet) written to by the loop. Similarly, we define the *output* of a loop to be all *last* memory stores, i.e. the last copy of all memory locations written by the loop body. The loop input and output are restricted to memory locations that are tainted. Figure 8 illustrates the notion of loop input and output.

In the case of nested loops, the input of an inner loop is copied to the input of the outer loop once the execution of the inner loop is finished. The analog is done for the inner loop output. There is one exception, namely if the memory input (or output) of the inner loop is not accessible to the outer loop. This happens if the inner and outer loop are in two separate functions and the inner loop input (or output) is stored on its local stack. Since our loop detection algorithm crosses function calls, this may happen. We simply do not copy input (or output) in that case.

3.5.2 Entropy Measures

Given the definition above for loop input and output we redefine entropy decreasing loops as loops with input that have a higher entropy than their

C Code Example

```
int i, sum;
int a[] = { 5, 7, 17, 11, 2 };

// Taint content of array a.
TAINT(a, sizeof(int) * 5);

for (sum = 0, i = 0; i < 5; ++i) {
  sum += a[i]++;
}
```

Loop Input and Output after Execution

```
   Input  a:{ 5, 7, 17, 11, 2 }
  Output  a:{ 6, 8, 18, 12, 3 }, sum:42
```

Figure 8: Example illustrating the notion of loop input and loop output.

output. The input and output can be seen as two memory buffers. To detect decreased entropy, we compute the entropy on both of these buffers and compare the results.

In our context it is crucial that the entropy values for buffers of different lengths be comparable. We strive for a length independent entropy measure. For example, if the loop input is 10 bytes long and the output is 20 bytes long, we require that their entropy values can be compared as if the input and output were of the same length. We therefore scale our entropy measures to a value between zero and one.

Information entropy is usually defined as follows:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \cdot log_2\ p(x_i), \tag{1}$$

where X is a discrete random variable and $p(x_i)$ the probability of the ith element of X. Information entropy is always defined over the alphabet of the random variable. Instead of changing the alphabet to match that of every buffer, we always use the 256 possible bytes as the alphabet and scale the entropy by dividing the result by its upper bound. Keeping a constant alphabet results in a better measure for comparing buffers with different sizes. Given a buffer b of length n we define the scaled entropy of b as follows:

$$H(b) = \frac{-\sum_{i=1}^{256} \frac{|b|_i}{n} \cdot log_2\ \frac{|b|_i}{n}}{log_2\ \big(min(n, 256)\big)}, \tag{2}$$

where $|b|_i$ is the number of occurrences of the ith byte in the buffer b. For buffers larger than 256 bytes $H(X)$ will always lie between 0 and

$8 = log_2\ 256$. For buffers of length $n < 256$ bytes, $H(X)$ lies between 0 and $log_2\ n$.

We introduce two additional entropy measures to distinguish encrypted data from decrypted data for small buffers: the *number of unique* bytes within buffer b, and the *number of different* bytes within b. For clarity, consider the following list of integers: $(1, 7, 1, 5, 7)$. The number of unique integers is 1 (5) whereas the number of different integers is 3 $(1, 5, 7)$. For buffers larger than 64 bytes, we compute the average number of unique and different bytes using a sliding window of 64 bytes. Therefore, we compute and can compare our three entropy measures on buffers of all sizes.

These three entropy measures are compared and evaluated in Section 5.1.

3.5.3 Decryption Loop Detection Algorithm

Given a binary, we use our tool to dynamically analyze and extract an annotated control flow graph. The offline detection algorithm considers annotated loops within the CFG, entropy measures of loop input and output, and basic block annotations indicating whether or not arithmetic or xors are used on tainted data. The detection algorithm searches for loops that decrease entropy for any of the three entropy measures by more than 15% and contain xor and arithmetic operations in their loop body. This rather simple algorithm is very effective in practice. The output of the detection algorithm is the location(s) of the decryption loop(s).

3.5.4 Retrieving Decrypted Input

With the location of the decrypted loops in hand, dynamic analysis is repeated. Whenever one of the decryption loops is executed, we dump its loop output, which is the low entropy version of the loop input. In some cases the dumped data contains more than the decrypted data, such as the values of temporary variables. We group dumped data from multiple loop executions according to the memory location of the data. This technique isolates decrypted data from temporary variables and helps the human read the dumped data.

3.6 Big Picture

We have now completed a discussion of the design and architecture of our approach illustrated in Figure 1. In summary, the binary under consideration is dynamically analyzed a first time to generate an annotated control flow graph. The offline detection algorithm described above is then executed on the extracted control flow graph to locate the decryption loops and output their locations. Dynamic analysis is then repeated with knowledge of the location of the decryption loops, permitting retrieval of the decrypted input.

4 Implementation

Here we describe the implementation of the design presented in the previous section.

4.1 Approach

There are at least two different approaches to dynamic binary analysis: binary instrumentation and whole-system instrumentation. Binary instrumentation operates on a single process and possibly its child processes. The whole-system approach works at the machine emulator level, instrumenting the entire machine including the operating system.

In the context of our research we are interested in analyzing a single binary. To accomplish this task, binary instrumentation is simpler to work with than whole-system instrumentation. With the whole-system approach, one has to identify which binary and process is running in order to focus the instrumentation and analysis only on the binary of interest. This difficulty is completely avoided with binary instrumentation. An advantage of the whole-system approach is that it instruments the operating system kernel as well as user level code. Decryption may occur within the kernel, in which case we would not be able to detect it with binary instrumentation. However, if malware uses known kernel cryptographic libraries there are easier ways than the approach developed here to decrypt encrypted input received by malware, e.g. by wrapping API calls.

We decided to use binary instrumentation for the implementation of our tool, because it allows us to focus on the problem at hand without dealing with the technicalities inherent to the whole-system approach.

4.2 Instrumentation Framework

For the implementation of our analysis tool we use Valgrind, an open source instrumentation framework for building dynamic analysis tools [15] [17]. Valgrind has been used before for dynamic malware analysis and implementation of memory tainting [18]. In Valgrind, instrumentation is carried out on an intermediate representation (IR) that uses a RISC-like instruction set. This property makes the implementation of taint-tracking and monitoring of memory access patterns easier to implement with Valgrind than with other instrumentation frameworks (i.e., Pin [13]) because there is only one load and one store operation. The disadvantage of Valgrind is that it does not run on Windows which makes the evaluation of real malware samples more challenging. Our current implementation resulted from exploratory work and was not designed as a production tool. Future implementations of our design will be ported to Windows.

4.3 Valgrind

Valgrind uses a dynamic binary re-compilation approach: it converts the client program into an intermediate representation (IR) that gets instrumented with analysis code by a tool plug-in and then converts the instrumented code back into machine code. Valgrind translates code blocks on demand following the execution. Code blocks are cached after translation to speed up rerun if necessary. Each code block is instrumented independently. At a high level, the translation of a single code block is a three step process:

Phase 1. Disassembly: *machine code* → *IR*. The first step converts the machine code of the code block into its intermediate representation. Each x86 instruction is converted into one or more IR statements. At this point the translated IR passes through an optimization phase that removes redundant operations.

Phase 2. Instrumentation: *IR* → *IR'*. The second phase consists of instrumenting the IR with the analysis code. The IR of the code block can be arbitrarily modified, e.g. new statements can be added and/or existing ones can be removed.

Phase 3. Assembly: *IR'* → *machine code*. The last step is responsible for converting the instrumented code block back to x86 machine code so that it can be executed. This phase includes converting the IR to a list of instructions, performing register reallocation and converting the list of instructions to actual machine code.

The first and third phase are performed by the Valgrind core. The second phase is performed by the Valgrind tool plug-in that we have implemented.

4.4 Our Valgrind Plug-in

The implementation of our Valgrind plug-in is split into five modules: instrumentation, control flow graph, shadow memory, memory allocator and system call wrappers. These modules are not all independent of each other, but each performs a specific task. The next sections give a description of each of these modules.

4.4.1 Shadow Memory

Data Structure The shadow memory stores the taint status of each addressable memory location. To ensure that the shadow memory consumes very little memory in practice we use a page-table-like data structure similar to *MemCheck* [23]. Our current implementation is designed for a 32-bit address space. The entire address space is split into 64K chunks, each one

64KB in size. The primary map (PM) is an array that holds a pointer for each one of these 64K chunks. If an entire chunk is not tainted, then its pointer in PM is NULL, otherwise it points to a secondary map (SM). The SM holds the taint value for all 64K addresses within a chunk. Each taint value is an unsigned integer that identifies the source of the tainted data, i.e. which system call produced the tainting. Figure 9 illustrates this data structure. Thanks to this data structure the shadow memory only uses a few MB of memory in practice.

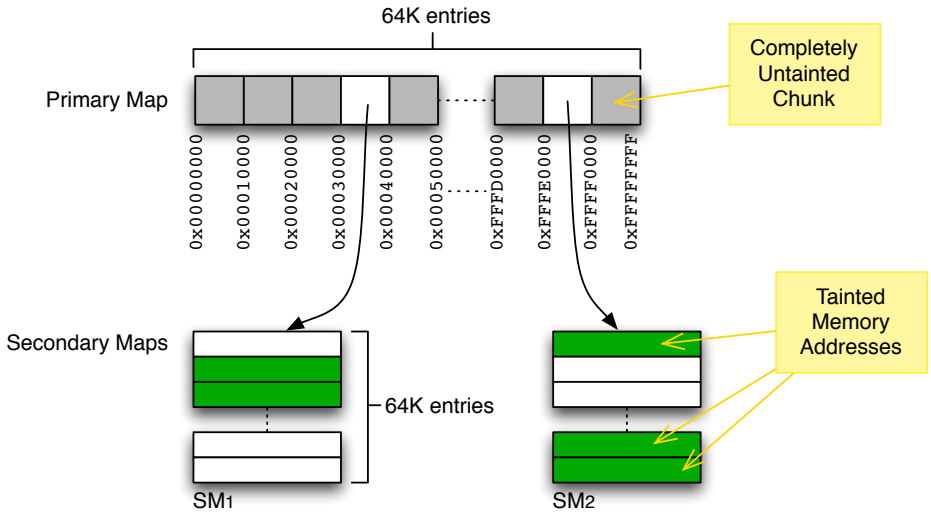

Figure 9: Structure of the shadow memory.

Operations At a high-level the operations provided by the shadow memory are `taint( Addr, SizeT )`, `untaint( Addr, SizeT )` and `istainted( Addr )`. For performance reasons additional methods are provided that implement special cases for these three operations, e.g. to taint two bytes at a time.

4.4.2 Instrumentation

The instrumentation module is by far the largest module of all. It implements the second phase in Valgrind's translation process and is responsible for instrumenting the translated code blocks. We instrument code blocks to track tainting, to detect loops and function calls, to track memory reads and writes during loop executions, to create basic blocks and the control flow graph, and finally, to set the instruction classes for each basic block.

Taint Tracking To implement the taint tracking we instrument every instruction that manipulates data and determine whether the result must be tainted or not. In Valgrind the intermediate representation has two kinds of instructions: statements and expressions. Statements represent operations with side-effects, such as stores and assignments. Expressions are operations without side-effects, e.g. arithmetic operations, loads and constants.

For each expression, we instrument the IR with code that computes whether or not the result of the expression must be tainted. The taint tracking policy is implemented as follows: the result of constant expressions (i.e., literal values) is never tainted. The result of a load is tainted if at least one of the loaded bytes is tainted *or* if the register that stores the address to load is tainted. The latter case is added to taint the result of table lookups where the table entry may not be tainted but its address is. Such table lookup operations appear frequently in cryptographic algorithms. For any other expression, the result is tainted if one of the operands is tainted, with some exceptions for constant functions such as `xor %eax %eax`.

The statements involved in taint tracking are register stores, memory stores and assignment to temporary variables. For each of these cases the destination is tainted if the right-hand-side (i.e. the result of the expression) is tainted.

Loop, Function and Basic Block Detection In order to dynamically detect loops, functions and basic block boundaries, we instrument all IR instructions that modify the program's control flow, such as conditional branches and jump instructions. Note: for conditional branches we also need to know if the branch is not taken, as this may indicate that a loop has finished an iteration.

Memory Reads and Writes Instrumentation of memory reads and writes is rendered easy in Valgrind thanks to its LOAD and STORE instructions. For every load and store executed within a loop body we keep track of the loaded (or stored) memory content. This information is used to compute the difference in entropy between loop input and output.

4.4.3 System Call Wrappers

Valgrind provides two callbacks that get executed before and after each system call. This functionality greatly simplifies system call wrapping. To implement the taint source described in Section 3.3.1, we provide a wrapping function for every system call that reads data from a file descriptor: `read()`, `socketcall()`, `recv()`, `recvfrom()` and `recvmsg()`. After the system call executes, we check whether the received data is read from a tainted file descriptor, and therefore has a tainted origin.

Note: we do this check at every read to properly handle non-blocking sockets. Future implementations will implement the complete file descriptor state machine to avoid checks at every read.

4.4.4 Memory Allocation and Deallocation

To reduce the false positive rate of the tainted memory, we untaint newly allocated and deallocated memory regions. Stack and heap memory allocation and deallocation functions are wrapped to perform this task. Valgrind provides functionality to replace standard C library functions related to allocation and deallocation of memory. The `brk` system call is wrapped as well.

4.4.5 Control Flow Graph

The control flow graph is dynamically constructed as the program executes. The instrumentation module informs the CFG module of changes in the control flow. We store tainting information, execution counters etc. for each basic block within the control flow graph.

At the instrumentation phase Valgrind provides a super block (SB) which is a single-entry, multiple-exit block of code. In comparison basic blocks (BB) are single-entry, single-exit blocks of code. Because the same BB may occur in multiple SBs it may happen that a BB gets split into two BBs. Figure 10 shows an example of a BB split. Since we collect information regarding each BB, spitting a BB is problematic. To cope with this problem we reduce the maximum size of SBs to one instruction, resulting in each SB having exactly one BB. This modification negatively impacts performance. As it turns out, we need this modification for another reason, namely to handle self-modifying code. Valgrind does not handle self-modifying SBs; if a SB modifies *itself* Valgrind does not retranslate it. Reducing the size of SBs to one instruction circumvents this issue. In order to analyze malware, we need to properly handle self-modifying code.

At the end of the analysis phase we reduce the size of the control flow graph by collapsing as many adjacent basic blocks as possible and aggregating the collected information for each collapsed BB. Our tool finally outputs the annotated control flow graph as a Python script for further offline analysis.

4.5 Detection Algorithm

The first stage of the analysis produces an annotated control flow graph that is output as a directed graph in Python. We decided to use Python to facilitate modifications of the detection algorithms. Since the detection algorithm is running offline, performance constraints are reduced. The decryption loop detection algorithm is described in Section 3.5.3. The output

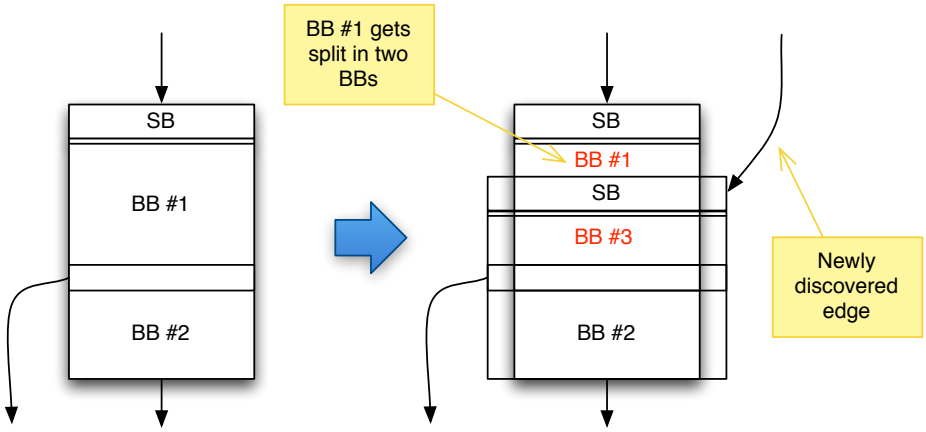

Figure 10: Example of a basic block split.

of the detection algorithm is a list of loop target addresses that correspond to the decryption loop.

4.6 Optimizations

We focused our implementation efforts more on effectiveness than performance since our analysis is typically done offline and is not usually performance critical. Nevertheless, binary instrumentation is very expensive and requires some optimization to execute reasonably fast. Particular care was taken in the implementation of the shadow memory (as proposed by Nethercote and Seward in[16]) and in the instrumentation responsible for taint tracking because both involve almost every single instruction.

During our case study of real malware we encountered performance challenges. The malware we analyzed starts by unpacking itself which takes minutes in our current infrastructure. Once unpacked, the malware is busy collecting information about the infected host. It is only after a couple minutes that the malware starts communicating over its encrypted command and control channel. To reduce the analysis time of this early behavior that is not related to the decryption process, we do not instrument the binary before any encrypted data is received. In other words, we start instrumenting the binary after the first memory bytes get tainted. Unfortunately, we cannot simply start instrumenting the binary when it receives its first encrypted bytes because we may not instrument crucial parts of previously executed code already stored in the translation cache. To deal with this issue we extended the Valgrind core to support clearing of the translation cache. As soon as the first tainted data is read, we clear the entire translation cache and start instrumenting code blocks with the analysis code to

extract the various features. With this technique the first stage of the malware's execution does not get instrumented and therefore executes much faster.

5 Experimental Evaluation and Results

In the following section we discuss the evaluation of our design and implementation. Our evaluation consists of three main parts. First we evaluate the entropy metrics used to distinguish between encrypted and decrypted data. The second part evaluates the effectiveness and the performance of our system through analysis of various Linux cryptographic tools and libraries. Finally, we present a case study of real malware that uses encrypted network traffic. All experiments were run on a Linux machine with dual core Intel CPU and 4GB of RAM.

5.1 Entropy Metrics

We introduced three different ways to compute the entropy of a buffer in Section 3.5.2: the scaled information entropy, the number of unique bytes and the number of different bytes. Here we compare the effectiveness of these measures for different buffer lengths.

To compare these measure we encrypt a text file (RFC 3268) with GnuPG and use our three metrics to compare its entropy with the entropy of the decrypted text file. The figures below show the entropy measurements for different buffer lengths. For a given buffer length, the entropy is the average entropy over a sliding window of that length.

For large memory buffers the scaled information entropy of encrypted data is clearly higher than for decrypted data. Figure 11 shows how scaled information entropy changes with increasing buffer size. Notice the bump in the graph when the buffer length equals 256 bytes. For buffers of length $n \leq 256$ bytes the entropy is scaled with $log_2(n)$ because the number of different characters is bound by the length of the buffer rather than the size of the alphabet. For buffers of length $n \geq 256$ our measure is scaled with $log_2(256) = 8.0$, corresponding to the traditional information entropy measure. If we instead always divide by 8.0 we reduce our ability to compare the entropy of buffers of different lengths because small buffers will always appear to have lower entropy than larger buffers (since $log_2(n) \leq log_2(256)$).

For memory buffers with a size smaller than 32 bytes, our scaled entropy measure is not very good at distinguishing between the encrypted and decrypted data. For this reason we use two additional entropy measures that make this distinction better for small buffers: the *number of unique* bytes and the *number of different* bytes. Figure 12 shows how the difference in entropy between encrypted and decrypted data is larger for these two measures than for the scaled entropy.

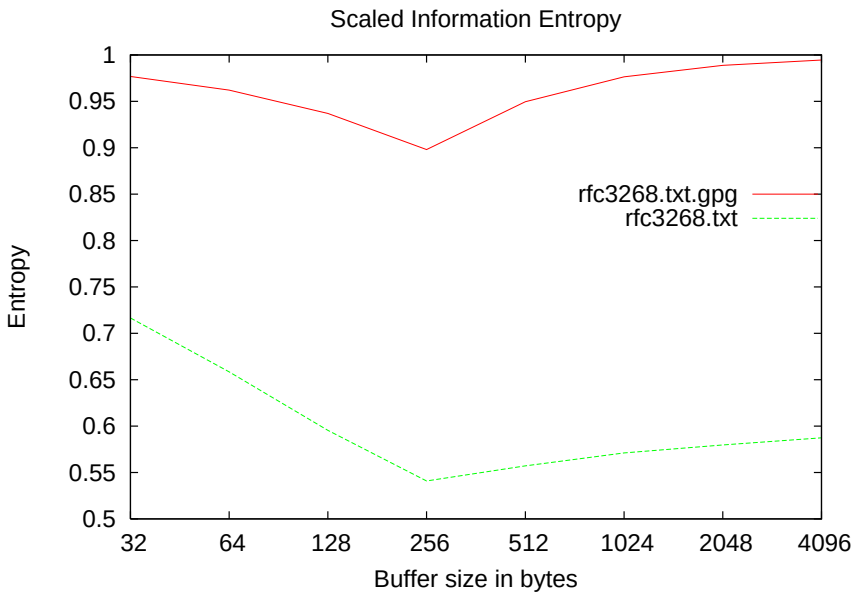

Figure 11: Scaled information entropy for large buffer sizes.

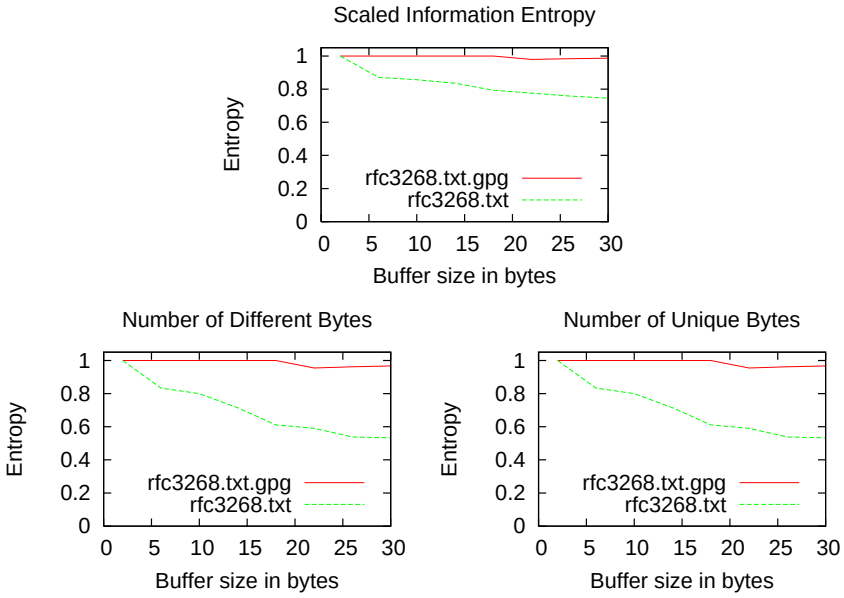

Figure 12: Compare three different entropy measure on small buffers.

5.2 Effectiveness and Performance

To measure the effectiveness of our design and implementation we analyzed different Linux binaries that use common cryptographic algorithms. Each of these binaries was analyzed separately with our analysis tool.

As far as we know, aside from a few exceptions, malware authors do not currently use well-known cryptographic algorithms. Instead, for the most part, authors use custom encryption algorithms. We believe that as soon as their economic model is threatened by security researchers who break their simple cryptographic algorithms, malware authors will shift to using secure cryptographic algorithms. Therefore, our choice of common cryptographic algorithms for the evaluation of our tool is justified and necessary in order to demonstrate that our approach is generic.

5.2.1 Walk Through Example: libgcrypt and AES

Here we walk through the analysis of libgcrypt decrypting a file that is encrypted with AES. We wrote a short program that reads an encrypted text file and decrypts it using libgcrypt. The encrypted file is only 46 bytes in size to make the analysis easier to follow. This file is the taint source. After the first analysis step our system produces an annotated control flow graph of the entire execution containing 5290 vertices. Upon execution the decryption loop detection algorithm finds three candidate loops:

```
Name:    cipher.c:961 (0x080552B5)
Type:    OutSwipe Swipe2
Exec:    2
Iter:    16 / 16 / 32
#In/Out: Loads:  Own: 16 / 32 / 48, Child: -
         Stores: Own: 36 / 36 / 72, Child: -
In/Out:  Loads:  U:0.78 / H:0.95 / D:0.89 / P:0.34
         Stores: U:0.57 / H:0.88 / D:0.74 / P:0.58
         Diff:   U:0.21 / H:0.07 / D:0.15 / P:-0.24

Name:    cipher.c:975 (0x0805537C)
Type:    InOutSwipe Swipe2
Exec:    1
Iter:    14 / 14 / 14
#In/Out: Loads:  Own: 28 / 28 / 28, Child: -
         Stores: Own: 32 / 32 / 32, Child: -
In/Out:  Loads:  U:0.93 / H:0.99 / D:0.96 / P:0.43
         Stores: U:0.59 / H:0.91 / D:0.78 / P:0.62
         Diff:   U:0.33 / H:0.08 / D:0.18 / P:-0.20

Name:    cipher.c:957 (0x08055265)
Type:    OutSwipe
Exec:    1
Iter:    2 / 2 / 2
```

```
#In/Out: Loads:  Own: 32 / 32 / 32, Child: 32 / 32 / 32
         Stores: Own: 116 / 116 / 116, Child: 116 / 116 / 116
In/Out:  Loads:  U:0.75 / H:0.95 / D:0.87 / P:0.31
         Stores: U:0.61 / H:0.87 / D:0.75 / P:0.53
         Diff:   U:0.14 / H:0.08 / D:0.12 / P:-0.21
```

Continuing with this example, below we reproduce the source code of the first loop ($0x080552B5$) which is part of the cipher feedback block (CFB) chaining mode implemented in libgcrypt. The loop xors the encrypted initialization vector (`c->iv`) with the encrypted input (`inbuf`) which results in the decrypted input (`outbuf`). Notice how the CFB mode encrypts the initialization vector even during decryption. The complete code of the decryption method that contains all three loops is reproduced in Appendix A.

```
956   /* encrypt the IV (and save the current one) */
957   memcpy( c->lastiv, c->iv, blocksize );
958   c->cipher->encrypt ( &c->context.c, c->iv, c->iv );
959   /* XOR the input with the IV and store input into IV */
960   for(ivp=c->iv,i=0; i < blocksize; i++ ) {
961       temp = *inbuf++;
962       *outbuf++ = *ivp ^ temp;
963       *ivp++ = temp;
964   }
```

We now describe the format of output loops using the first loop as an example.

`Name` The name of the source file and line number are extracted from the debugging symbols and used only for verification purposes. Debugging symbols are not used in any way by the detection algorithm. The address of the loop target is what normally comprises the `Name` of the loop.

`Type` indicates the detected loop type. In the case of the first loop, the type field indicates that the loop swipes over a buffer and writes tainted data block by block using a block size that is a power of two.

`Exec` shows the number of loop executions. The first loop is executed once for every complete 16 byte block of the encrypted input (since we are using 128-Bit AES). Remaining bytes are decrypted in the second loop ($0x0805537C$). Since the input is 46 bytes long and the block size is 16 bytes the first loop is executed twice.

`Iter` indicates the minimum, maximum and total number of loop iterations over all executions. We see that the first loop always iterates a constant number of times, namely 16 times which corresponds to the block size.

`#In/Out` holds the minimum, maximum and total number of tainted bytes that were loaded or stored by the loop. This information is aggregated over multiple executions. During a loop execution only unique load and store addresses are counted. For space and performance reasons we do not keep track of whether a loaded (or stored) address is the same over multiple executions.

Note that for the first loop the minimum number of loads is 16 bytes and the maximum is 32 bytes. These numbers reflect that during the first execution of the loop the initialization vector `c->iv` (IV) is not yet tainted. During the second iteration, however, the IV is tainted (because it is set to the first encrypted block) and therefore 32 tainted bytes are loaded. The 36 stored tainted bytes correspond to the 16 bytes of the decrypted buffer plus the 16 bytes from the IV plus the 4 bytes that store the temporary variable `temp`.

`In/Out` shows the change in entropy between the loop input and output, averaged over all loop executions. `U`, `H`, `D`, and `P` stand for number of unique bytes, scaled entropy, number of different bytes and printability respectively. Printability is an additional measure representing the percentage of printable characters within the buffer.

Notice how in the case of the first loop, the number of unique bytes and the number of different bytes distinguish better between the entropy of the loop's input and output than the scaled entropy (0.21 and 0.15 versus 0.07). This case is a good illustration of how for small buffers scaled entropy is not as good at distinguishing between encrypted and decrypted data as our other two entropy measures.

The output of the detection algorithm for the second and the third loops in our example contains similar information. Looking at the source code in Appendix A, one can see that the output of the first and the last loop are identical because the first loop is an inner loop of the third. The second loop handles the final, incomplete block if there is any.

After the detection of these three loops we run our analysis a second time with information regarding the loops' locations. The second analysis process collects the tainted output of these three loops. Duplicated values are removed, and memory that is located together is grouped. Finally, the different buffers at the different locations are output. For each buffer we compute the entropy again to help the analyst that runs the tool to identify the decrypted input. In some cases temporary variables are also output but can easily be spotted either by looking at their entropy or by the fact that they are not located next to a buffer.

5.2.2 Additional Evaluations

Blowfish and Twofish In order to substantiate the evidence for our tool's effectiveness we investigated two additional cryptographic algorithms and implementations: blowfish and twofish. For these measurements we encrypt and decrypt a text file (RFC 3268) with open source Linux file encryption utilities that use these algorithms (Bcrypt and Twofish). The encrypted text file is the taint source.

In both of these cases we were able to successfully locate the decryption loops and the decrypted file content in the program's memory.

OpenSSL We extended the evaluation of our tool with the analysis of OpenSSL. The command-line tool cURL is used to fetch a URL over an encrypted HTTP connection using SSL. The socket that is connected to the IP of the web server that hosts the URL is defined as the taint source in this case.

In the first experiment we fetch a website from a local Apache server that uses AES for its symmetric cipher: `curl -k https://localhost`. Note: the local web server does not have a valid certificate which is why the `-k` parameter is added to explicitly skip the certificate verification. The second experiment calls `curl https://mail.google.com`. Google's web servers use RC4 as the symmetric cipher. For both of these cases our system was able to identify the location of the decrypted webpage in memory.

The third experiment investigates the ability of our system to analyze a Python script that fetches the same URL as before. The analyzed Python script looks like this:

```
#!/usr/bin/python2.4
import urllib2

f = urllib2.urlopen("https://mail.google.com")
print f.read()
f.close()
```

Even for interpreted code our automatic analysis tool is able to detect the decryption loop and the location of the decrypted webpage in memory. Note that the analysis is much slower because the entire Python interpreter gets instrumented.

GnuPG For the last experiments we investigated GnuPG, an open source implementation of PGP. We analyze GnuPG as it decrypts an encrypted text file (same RFC as used above). The file is declared as being the taint source.

We considered two scenarios, one in which we used GnuPG's compression functionality to compress the text file prior to encryption and another in

which we did not use compression. GnuPG uses libz as its compression library. If compression is used, the encrypted file is first decrypted and then decompressed.

The analysis of GnuPG without compression successfully detects the decryption loop. When compression is used, our tool detects the decompression loop (libz's inflate loop) instead of the decryption loop, as the entropy of compressed data decreases greatly upon decompression. Since the entropy of encrypted data and compressed data are both very high, the actual decryption loop is not detected by our heuristic.

In this example, the detection of the decompression loop is actually what we want to achieve, since our analysis aims to extract the de-obfuscated version of the input. In this case, compression can be interpreted as a kind of obfuscation. At the same time, this result shows the limitation of our heuristic that relies on the assumption that the decryption process decreases memory entropy. For example, imagine the case in which a binary update is received over an encrypted communication channel. If packed, the binary itself will have high entropy. Our tool would fail to detect the decryption process in this case since decryption would not significantly decrease entropy. To cope with this limitation, future work will search for additional heuristics to distinguish encrypted from decrypted data.

5.2.3 Performance

For every experiment mentioned above we measure the time required to do the analysis and to extract the annotated control flow graph. Table 4 shows the impact of our analysis on the execution time of the analyzed binaries. For the experiments above our system slows the binaries' execution by a factor of 2400 on average.

This effect on the binary's execution time seems very large. The goal of our implementation was to demonstrate the effectiveness of our design, not to maximize performance. It is important to keep in mind that our analysis extracts the program's entire control flow graph and therefore many more features than are actually used in the final decryption loop detection algorithm. Future work will focus on implementing our detection algorithm in a single analysis process and on improving performance.

5.3 Case Study: Kraken

As a case study we analyzed the Kraken bot which has been getting a lot of press attention lately. Kraken is a Windows bot binary that uses an encrypted command and control (C&C) channel to communicate back to its bot master. This malware is particularly interesting to us because it uses a custom symmetric encryption algorithm to encrypt its communication and control channel. Appendix B describes the bot's function in more detail.

Table 4: Performance impact of our current implementation on binary's execution time, compares analysis time of our system with the normal execution time.

Experiment	Execution + Analysis	Normal Execution	Factor
Bcrypt (blowfish)	1.47s	1ms	1470
Twofish	1.69s	1ms	1690
Libgcrypt	6s	3ms	2000
cURL (AES)	119s	25ms	4760
cURL (RC4)	46s	14ms	3286
Python (OpenSSL)	247s	87ms	2839
GnuPG (w/ libz)	120s	75ms	1600
GnuPG (w/o libz)	118s	73ms	1616

5.3.1 Evaluation Setting

Unfortunately Valgrind does not run on Windows. In order to use our Valgrind tool to analyze Kraken, we use Wine. Wine is an open source implementation of the Windows API and is intended as a compatibility layer for running Windows programs in Linux. Therefore, instead of analyzing Kraken directly we analyze Wine which is running Kraken.

When dynamically analyzing malware, one must take care to not negatively impact Internet users. For example, we wouldn't want the malware under analysis to begin sending spam. On the other hand, we can't block all network traffic because we want to be able to intercept and decrypt any encrypted input that the malware receives. To handle this difficulty, we sandbox the malware's execution using a virtual machine (VM) that selectively allows network traffic. The virtual machine setting we use was presented previously by Provos, Polychronakis et al. [22] [20]. The execution environment of Kraken involves the virtual machine running Linux that executes Valgrind, which analyzes Kraken running in Wine. Essentially, the execution environment looks like this: VM → Linux → Valgrind → Wine → Kraken.

5.3.2 Collecting Interesting Malware Binaries

Collecting malware binaries that receive encrypted network traffic and are therefore interesting for us to analyze with our tool is a challenging task.

One way to solve this problem is to manually search for these binaries and observe their use of encrypted traffic. This approach was used to find the Kraken bot, which we received from another security researcher. The manual approach to finding interesting malware is slow and does not scale. We

therefore investigated an automatic approach to detecting malware binaries that receive encrypted traffic.

Google's Safe Browsing team visits thousands of websites every day with virtual machines looking for malicious websites that host drive-by downloads [22]. These virtual machines block almost all network traffic except for HTTP traffic that is required for the virtual machine to get infected in the first place. Our idea is to search for encrypted network traffic within the HTTP communications generated by infected virtual machines. In particular, we are interested in the encrypted HTTP traffic that is received or sent by processes other than the browser. Presumably these other processes are malware binaries that use encrypted network traffic, since we do not expect any process other than the browser to communicate with the Internet.

The fact that we only have access to HTTP traffic limits our coverage of malware that uses encryption, since malware could use other non-HTTP protocols. This limitation is voluntarily imposed on our system in order to avoid negatively impacting Internet users.

To detect encrypted traffic we compute various measures on the HTTP payloads received by processes other than the browser. Measures include entropy, floating frequency, printability, detection of Windows PE binaries, randomness test etc.. To avoid misclassification we detect the MIME type of the received payloads. We found a fair number of instances where very high entropy content whose MIME type could not be detected was downloaded by the infected machine. We consider these high entropy payloads of unknown MIME type to be encrypted traffic. Our current infrastructure does not allow us to identify which binary, or even process, generated the encrypted payload. An infected machine typically downloads a dozen or so malicious binaries upon infection, any one of which could have received the encrypted traffic. We developed a semi-automatic infrastructure that extracts all downloaded binaries from the virtual machine and executes each one independently, hoping to find the binary that received the encrypted network traffic. If found, this binary would be a candidate binary for analysis with our encryption tool.

We evaluated about one hundred cases in which encrypted network traffic was received. For each of these cases we ran all of the binaries in our Wine infrastructure to find the binary that received the encrypted network traffic. Unfortunately our analysis did not find any binary that runs in our infrastructure and fetches the encrypted input. A lot of malware binaries cannot be executed in Wine. In addition, the dependencies between all downloaded malicious binaries are unknown. We simply run each binary individually, which may not reproduce the necessary running environment. We leave it for future work to fully automate this process and resolve these infrastructure issues.

5.3.3 Infrastructure Challenges

The Kraken bot version 3.16 executes smoothly in Wine. When we analyzed Kraken, it's bot master had moved and was no longer reachable. All that is necessary for our analysis is to be able to send an encrypted payload to the Kraken bot and analyze it as it decrypts the received payload. We looked online for Kraken network traces that we could replay.

The first step towards replaying the traffic was to create a fake DNS service to answer the Kraken DNS queries. In its first stage, Kraken tries to locate its bot master by requesting the IP addresses of some random domain names registered at free top level domains like dyndns.org. Our fake DNS answers all of these queries with a bogus IP address to lead the Kraken bot to believe that its bot master is still alive.

Once the bot knows its bot master's IP address, it tries to connect to this IP on a random UDP port. We have no way of predicting this random port. We modify the virtual machine's network stack implementation to add a UDP sink that makes it possible to route all UDP ports but a few to a single IP and port. This technique allows us to route all UDP traffic from Kraken to our simulated bot master running outside of the virtual machine.

Every Kraken bot uses a different encryption key that depends upon the infected machine's hardware. Furthermore, Kraken expects the bot master to use an encryption key that depends upon its key. Unfortunately, these constraints mean that the network traffic found online cannot be replayed since the encryption keys use by the bot master will most certainly be different. Kraken simply discards any encrypted traffic encrypted with the wrong key. To solve this problem, we use the previously published algorithm that is able to decrypt Kraken traffic to decrypt the network traces that we collected [11]. We then re-encrypt the bot master's responses with the appropriate key.

To accomplish this re-encryption, we reverse engineered the encryption algorithm from the decryption algorithm and found the dependency between the bot master and bot key by debugging the Kraken bot. All of this work results in a bot master that sits outside of the virtual machine and is able to replay previously collected Kraken traffic by re-encrypting it on the fly for whatever Kraken bot is running inside the virtual machine. We are confident that our replay technique works because Kraken replies to our replayed and re-encrypted payloads.

The final and biggest challenge we faced occurred when analyzing Kraken and Wine in Valgrind. Kraken employs rarely used x86 machine instructions that are not implemented in Valgrind's emulator. We added the support for about half a dozen additional instructions to Valgrind. Finally, Kraken also uses a wrongly encoded instruction that does not comply with the Intel manuals, but which runs smoothly on real hardware. Since Valgrind is implemented according to the Intel manuals, this instruction did not execute

in Valgrind. We therefore mimicked the hardware's behavior by implementing this bug into Valgrind's emulator. We believe that Kraken uses these instructions to detect and evade emulation.

5.3.4 Measurements and Results

After dealing with all of the infrastructure challenges mentioned in the previous section, we were able to analyze the Kraken bot as it decrypts the encrypted network traffic that it receives. We successfully detected the entropy decreasing loop and extracted the decrypted payload from the memory.

When analyzing Kraken with the virtual machine, Valgrind, and Wine, we experienced a huge performance overhead. The technique used to reduce this overhead was discussed in section 4.6. Obviously, the current infrastructure with its multiple emulation layers is not a fair way to measure the performance of our approach.

6 Limitations and Future Work

Our heuristic assumes that decryption is performed in a loop and that encrypted data has higher entropy than decrypted data. As mentioned earlier in this report, some simple substitution ciphers do not affect information entropy and would therefore not be detected using our heuristic. Our heuristic would also fail to locate the decrypted input if it has high entropy. As long as one can distinguish between encrypted and decrypted input, our tool can find the decryption loops. Our heuristic merely states that entropy is a good measure for making this distinction. For every case in which this assumption does not hold true, one could substitute another superior measure to distinguish encrypted data from decrypted data.

Our approach is based on dynamic binary analysis. The greatest limitation of this type of analysis is that it can be detected and evaded, rendering the analysis useless. This issue may be solved for a single malicious binary, as demonstrated in the evaluation section, but it becomes especially challenging when trying to automatically analyze a large number of binaries, which is our ultimate goal. In our research we use Valgrind, which is prone to detection because it sits in the same process as the malware. In addition, the performance of our tool at this point is sub-optimal because we use several emulation layers. One clear consequence of this reduced performance is that it makes our analysis even easier to detect. In the next phase of this project, our goal is to move to an infrastructure that is more performant. Ferrie shows that even when using virtual machines like VMWare or QEMU, there are ways for malware to detect that it is being analyzed [6]. Future work will need to consider these challenges of detection and evasion.

Our design uses dynamic tainting techniques to track the memory that depends upon the encrypted input of the program. This technique effectively

reduces the number of candidate memory locations that may contain the program's decrypted input. Cavallaro et al. present practical examples of code that evades traditional taint tracking implementations [12]. At this point we do not provide a solution to this issue. To the best of our knowledge no malware currently uses these evasion techniques. Should use of these techniques become prevalent, future work will either address these detection challenges or replace taint tracking with alternative means to reduce the search space of memory.

In addition to addressing the limitations listed above, future work will tackle the infrastructure challenges brought to light in the evaluation section. Our tool's compatibility shortcomings prevent us from achieving reasonable coverage of malware that uses encryption, since only a small fraction of malware runs successfully in Wine. Therefore, our implementation must be ported to Windows in order to extend our tool to analyze a larger number of binaries. To achieve satisfactory coverage and scale our analysis to thousands of binaries, we need to complete our work to automate the collection of malware binaries that use encryption.

Future work will also strive to identify the decryption algorithm used by the binary under analysis, if it is among known cryptographic algorithms. For custom algorithms, we would like to detect the use of additional structures common to cryptographic algorithms, such as a Feistel network. Finally, we are interested in pursuing the extraction of any kind of cryptographic key material that is used.

7 Conclusions

More and more aspects of our daily lives depend upon the proper functioning of computers and the Internet. As a result, the consequences of an infected machine are increasingly detrimental to people's privacy and economic welfare. The use of encrypted communication has seriously impacted the ability to automatically analyze malware behavior, and therefore sets the security research community a step behind in the arms race against malware authors.

Existing approaches to decrypting encrypted network communication rely on manual analysis, which is both time and resource consuming and therefore does not scale. In this report, we propose a generic tool for the automatic decryption of encrypted input received by malicious binaries. Our approach uses dynamic analysis of malware to extract various features. We design a detection algorithm to search through these features and find the location of the decryption routine and the decrypted network traffic in the system's memory. Our approach builds on the fact that most decryption algorithm implementations include loops that decrease the information entropy of memory.

We demonstrate the ability of our tool to automatically locate the de-

crypted input from programs using well known cryptographically secure algorithms. In addition, we provide a case study in which we show the effectiveness of our tool in decrypting the Kraken bot's encrypted network traffic. The fact that our solution is generic and not tailored to a particular cryptographic algorithm is a clear advantage over existing solutions for the analysis of continuously changing malware. Several of the limitations of this work, and in particular performance, result from the exploratory nature of the research in its current status. As we adapt our tool for production, we will focus on resolving these shortcomings. We argue that our approach can be used for revealing the intent of attackers that hide their activities using encrypted network communication.

8 Acknowledgements

I thank Professor Bernhard Plattner for agreeing to mentor me and support my rather unusual thesis arrangement. I am grateful to Niels Provos for hosting me within his team during my thesis project and for his *killer advice* during rough passages. I am thankful to Google and the Anti-Malware team for giving me exposure to exciting work in the domain of security. In particular I would also like to thank Bernhard Tellenbach for his consistent dedication to my progress and for his true interest in my project. Finally, I would like to express my appreciation to Panayiotis Mavrommatis, Thomas Dübendorfer, Moheeb Rajab, Xin Zhao, Michalis Polychronakis, Steven Hanna, the entire Google Anti-Malware team and Amanda for idea bouncing, support and great feedback on my presentation and write up.

9 Appendices

A Libgcrypt Decryption Method

This code is copied from libgcrypt version 1.2.2. The source file is cipher.c located in the cipher folder. The code indentation was reduced to fit this page.

```
924 static void
925 do_cfb_decrypt( gcry_cipher_hd_t c, byte *outbuf,
926                 const byte *inbuf, unsigned int nbytes )
927 {
928   byte *ivp;
929   ulong temp;
930   size_t blocksize = c->cipher->blocksize;
931
932   if( nbytes <= c->unused ) {
933     /* Short enough to be encoded by the remaining XOR mask. */
934     /* XOR the input with the IV and store input into IV. */
935     for(ivp=c->iv+blocksize - c->unused; nbytes; nbytes--,c->unused--) {
936       temp = *inbuf++;
937       *outbuf++ = *ivp ^ temp;
938       *ivp++ = temp;
939     }
940     return;
941   }
942
943   if( c->unused ) {
944     /* XOR the input with the IV and store input into IV. */
945     nbytes -= c->unused;
946     for(ivp=c->iv+blocksize - c->unused; c->unused; c->unused-- ) {
947       temp = *inbuf++;
948       *outbuf++ = *ivp ^ temp;
949       *ivp++ = temp;
950     }
951   }
952
953   /* now we can process complete blocks */
954   while( nbytes >= blocksize ) {
955     int i;
956     /* encrypt the IV (and save the current one) */
957     memcpy( c->lastiv, c->iv, blocksize );
958     c->cipher->encrypt ( &c->context.c, c->iv, c->iv );
959     /* XOR the input with the IV and store input into IV */
960     for(ivp=c->iv,i=0; i < blocksize; i++ ) {
961       temp = *inbuf++;
962       *outbuf++ = *ivp ^ temp;
963       *ivp++ = temp;
964     }
965     nbytes -= blocksize;
```

```
966   }
967   if( nbytes ) { /* process the remaining bytes */
968     /* encrypt the IV (and save the current one) */
969     memcpy( c->lastiv, c->iv, blocksize );
970     c->cipher->encrypt ( &c->context.c, c->iv, c->iv );
971     c->unused = blocksize;
972     /* and apply the xor */
973     c->unused -= nbytes;
974     for(ivp=c->iv; nbytes; nbytes-- ) {
975       temp = *inbuf++;
976       *outbuf++ = *ivp ^ temp;
977       *ivp++ = temp;
978     }
979   }
980 }
```

B The Kraken Bot

The Kraken bot has recently been getting a lot of media attention. We became interested in analyzing this particular malware binary not only because it is currently relevant, but also because it uses an encrypted command and control (C&C) channel to communicate with its bot master. Kraken uses a custom block cipher encryption algorithm which makes it a perfect use case for our research.

The next sections describe various details about the Kraken bot.

B.1 Communication Protocol

In this section we describe the communication protocol used by Kraken version 3.16. As its transport layer protocol, Kraken uses UDP to send small packets, e.g. smaller than 500 bytes, and TCP for larger payloads. All packets sent and received over the C&C channel are of the same format. See Figure 13 for an illustration of the Kraken packet format.

Kraken uses a custom, deterministic block cipher encryption algorithm to encrypt the packet header and payload. The decryption keys are sent with every packet! This property renders the decryption of network traffic trivial if one knows the protocol and the decryption algorithm. We suspect that the malware author does not know any key exchange protocols and is sending the keys to make it possible for every bot to have a different key. To prevent signature-based network intrusion detection it is desirable that every bot have a different key if deterministic encryption is used. For more information about the decryption algorithm see Section B.2

B.2 Encryption Algorithm

Kraken's encryption algorithm was reverse engineered and first described by Ligh [11]. From the published code we analyzed the encryption algorithm further.

Kraken uses a 64-bit deterministic, block cipher with a 96-bit key. For messages that exceed 64 bits, Kraken simply partitions the message into 64-bit blocks and encrypts each separately. This mode is called electronic codebook or for short, ECB. The ECB mode of operation is the simplest but also the weakest mode of operation [1]. If the last block is shorter than 64 bits the remaining bytes are encrypted by xoring them with parts of the key.

The encryption algorithm works as follows: Let's assume a 96-bit key K, an n-bit plaintext message m that is split into t complete 64-bit blocks $m_1, ..., m_t$, and possibly one incomplete block m_{rem}. The ECB mode encrypts each m_i block as follows: $c_i = E_K(m_i)$. The block cipher function $E_K(\cdot)$ is described below. The incomplete block m_{rem} is encrypted byte-per-byte as follows:

for $i = 0$ to length of m_{rem} **do**
$c^i_{rem} = m^i_{rem} \oplus (K^i + K^{11-(i \bmod 4)})$,
end for

where K^i is the ith *byte* from key K.

The block cipher used by Kraken looks very much like a Feistel cipher. A

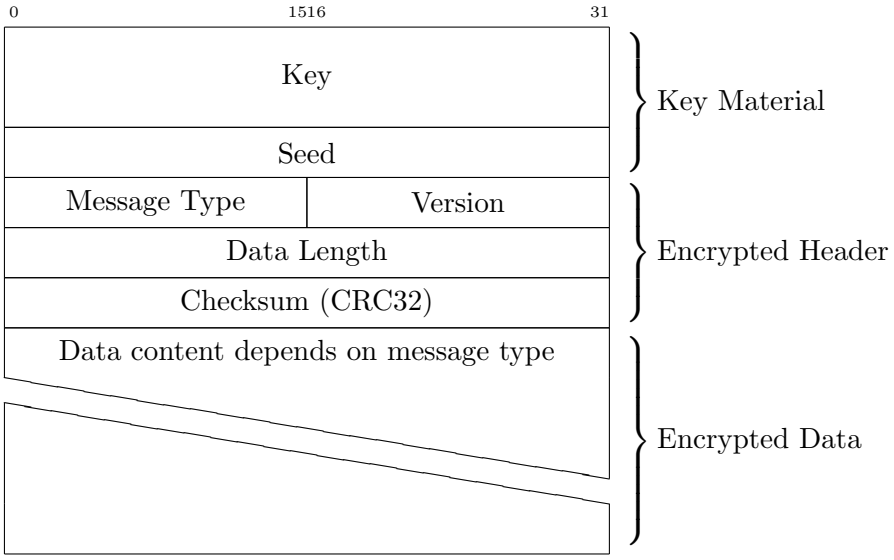

Figure 13: Kraken network packet format.

Feistel cipher is an iterative cipher that starts by splitting an n-bit plaintext block into two halves of length $n/2$: (L_0, R_0). At every iteration L_i and R_i get modified as follows:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i), \end{aligned}$$

where each subkey K_i is derived from the cipher key K. The interesting property of the Feistel network is that the f function need not be invertible to allow inversion, i.e. decryption, of the Feistel cipher. In fact because the xor is used to combine L_{i-1} and R_{i-1} the following is always true:

$$\big(L_{i-1} \oplus f(R_{i-1}, K_i)\big) \oplus f(R_{i-1}, K_i) = L_{i-1}.$$

The only difference between the Kraken cipher and a Feistel cipher is that the Kraken cipher uses subtraction and addition instead of xor to combine the left half with the output of the round function. For unsigned 32-bit integers this is also always true:

$$\big(L_{i-1} - f(R_{i-1}, K_i)\big) + f(R_{i-1}, K_i) = L_{i-1},$$

which makes subtraction and addition a working replacement for the xor operator.

For a 64-bit message m and a 96-bit key K split in three 32-bit keys K_1, K_2, K_3, the Kraken function f is defined as follows:

$$f(m, K_1, K_2, K_3) = (m \ll 4 + K_1) \oplus (m + K_3) \oplus (m \gg 5 + K_2).$$

It is interesting to note that malware authors do not bother to implement a more secure cipher block chaining mode that would be only slightly more complicated to implement.

B.3 Binary Packing

The Kraken binary is packed using a custom packer. We were not able to unpack the binary using common automatic unpackers. In fact none of the common packer detectors, such as PEiD, recognized the packing algorithm. We also used more sophisticated unpacking tools provided through online services: EUREKA! [29] and Renovo [8]. The result of these analysis tools is valuable for further manual analysis: the Kraken binary was at least partially unpacked. Unfortunately this is not satisfactory in our case, since we need to run the unpacked binary to analyze it.

None of the unpacked binaries exhibits the behavior we are interested in, namely the encrypted communication.

C Loop Type Detection Algorithms

In this section we describe the algorithms used to detect the loop types introduced in Section 3.4.3. See Algorithms 1, 2, 3 and 4. These algorithms all take a *LoopExecution* structure as a parameter that in this context stores the following attributes:

ustores contains minimum, maximum and total number of unique stored tainted memory locations during the loop's execution.

uloads contains the same as *ustores* but for unique loaded tainted memory locations.

numiter contains the number of loop iterations for the loop's execution.

Algorithm 1 IsAccumulatorLoop(LoopExecution l)

```
if l.ustores_min == 0 and l.ustores_max == l.ustore_total then
  return True
end if
return False
```

Algorithm 2 IsOutSwipeLoop(LoopExecution l)

```
d = l.ustores_max − l.ustores_min
if l.numiter > 1 and l.ustores_total > l.ustores_max and
l.ustores_min · l.numiter + d == l.ustores_total then
  return True
end if
return False
```

Algorithm 3 IsInOutSwipeLoop(LoopExecution l)

```
d = l.uloads_max − l.uloads_min
if IsOutSwipeLoop(l) and uloads_total > uloads_max and
uloads_min · l.numiter + d == l.uloads_total then
  return True
end if
return False
```

Algorithm 4 IsSwipe2Loop(LoopExecution l)

if $IsOutSwipeLoop(l)$ and $(l.ustores_{min}$ & $(l.ustores_{min}-1)) == 0$ **then**
  return True
end if
return False

References

[1] Scott A. Vanstone Alfred J. Menezes, Paul C. Van Oorschot. *Handbook ofApplied Cryptography*. CRC Press, 1997.

[2] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143, 2006.

[3] Ken Chiang and Levi Lloyd. A case study of the rustock rootkit and spam bot. In *HotBots, First Workshop on Hot Topics in Understanding Botnets*, April 2007.

[4] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46, May 2005.

[5] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, 2003.

[6] Peter Ferrie. Attacks on virtual machine emulators. Symantec Security Response, December 2006.

[7] Dirk Janssens, Ronny Bjones, and Joris Claessens. Keygrab t00 - the search for keys continues... Technical report, Utimaco Safeware AG and COSIC, 2000.

[8] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, New York, NY, USA, 2007. ACM.

[9] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 273–288, Berkeley, CA, USA, 2006. USENIX Association.

[10] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables, 2005.

[11] Michael Hale Ligh. Kraken encryption algorithm. Technical report, http://mnin.blogspot.com/2008/04/kraken-encryption-algorithm.html, 04 2008.

[12] R. Sekar Lorenzo Cavallaro, Prateek Saxena. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Stony Brook University, 2007.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[14] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec. 2007.

[15] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.

[16] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. *ACM Virtual Execution Environments*, 2007.

[17] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[18] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS 05)*, 2005.

[19] Cody Pierce. Owning kraken zombies, a detailed dissection. Technical report, http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies, 2008.

[20] Michalis Polychronakis, Panayiotis Mavrommatis, and Niels Provos. Ghost turns zombie: Exploring the life cycle of web-based malware. In *1st USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[21] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, April 2006.

[22] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. Technical report, Google Technical Report, 2008.

[23] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[24] Adi Shamir and Nicko van Someren. Playing hide and seek with stored keys. *Financial Cryptography*, pages 118–124, 1999.

[25] Sergei Shevchenko. Memory stealthiness of kraken. Technical report, http://blog.threatexpert.com/2008/05/memory-stealthiness-of-kraken.html, 2008.

[26] Peter Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[27] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 14, Washington, DC, USA, 1998. IEEE Computer Society.

[28] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[29] Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Monirul Sharif. Eureka: A framework for enabling static analysis on malware. Technical Report SRI-CSL-08-01, Computer Science Laboratory and College of Computing, Georgia Institute of Technology, April 2008.