Reuse-Oriented Camouflaging Trojan: Vulnerability Detection and Attack Construction*

Zhiqiang Lin (contact author) Xiangyu Zhang Dongyan Xu
Dept. of Computer Science and CERIAS, Purdue University, West Lafayette, IN 47907, USA
{zlin,xyzhang,dxu}@cs.purdue.edu, 765-494-6182, 765-494-0739(fax)

Abstract

We introduce the reuse-oriented camouflaging trojan – a new threat to legitimate software binaries. To perform a malicious action, such a trojan identifies and reuses an existing function in a legal binary program instead of implementing the function itself. Furthermore, this trojan is stealthy in that the malicious invocation of a targeted function usually takes place in a location where it is legal to do so, closely mimicking a legal invocation. At the network level, the victim binary can still follow its communication protocol without exhibiting any anomalous behavior. Meanwhile, many close-source shareware binaries are rich in functions that can be maliciously "reused", making them attractive targets of this type of attack. In this paper, we present a framework to determine if a given binary program is vulnerable to this attack and to construct a concrete trojan if so. Our experiments with a number of real-world software binaries demonstrate that the reuse-oriented camouflaging trojans are a real threat and vulnerabilities of this type in legal binaries can be effectively revealed and confirmed.

Keywords: Software reliability, trojan horse, binary analysis, software testing, binary code reuse

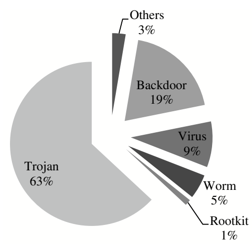
Category: DCCS – Regular Papers (approx. 7000 Words)

1 Introduction

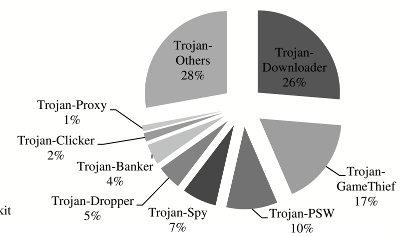
Trojan is a type of malware that appears to perform a desirable function for users but in fact contains malicious logics. It has been a major threat to software security and reliability. According to our study on malware samples in *VxHeaven* [4], trojans remain a dominant malware category. As shown in Fig. 1(a), trojans account for 63% of all 266980 pieces of malware, whereas the shares of virus, worms, and rootkits are 9%, 5%, and 1%, respectively. Another study by BitDefender [1] shows that, from January to June 2009, the trojan-type of malware is on the rise, accounting for 83% of the global malware detected in the wild.

Trojans have diverse payloads. They can be used: in botnets to perform malicious tasks such as SPAM or DDoS; for unauthorized download and installation of software (including other trojans), which accounts for 26% of the trojans in our study (shown in Fig. 1(b)); for data theft (e.g., game data theft (17%) and password stealing (10%)); for spying (7%); as droppers (5%); as bankers (4%); and as clickers (2%).

*The material in this paper has been cleared.



(a)



(b)

Figure 1. Distribution of (a) malware types and (b) trojan payloads in VxHeaven.

While most existing trojans are implemented as *new, independent pieces of code*, in this paper, we demonstrate that trojans can be more stealthily constructed by *reusing* functions from existing, third-party software binaries. We call such attacks on existing binaries *Reuse-Oriented Camouflaging* (or ROC for the rest of the paper) attacks. Moreover, we show that real-world software binaries may be vulnerable to ROC attacks and we define such vulnerability as the *ROC vulnerability*. We demonstrate that the detection of ROC vulnerabilities as well as the construction of ROC attacks (i.e., creation of ROC trojans for confirming the vulnerabilities) are not just feasible but can be made highly systematic.

The key observation behind ROC attacks is that certain functional features in legal software binaries can be subverted for malicious purposes. For example, an FTP program has all the basic capabilities to steal and transfer privacy-sensitive files; an email client has all the functions necessary to send spams. More specifically, with a ROC trojan for spamming, the subject and content of a spam message could be supplied to the proper mail-sending function, which will then send out the spam just like a regular email. The attacker does not have to perform any environment setup such as socket creation, handshaking, and payload encoding.

ROC trojans have unique properties: (1) Statically, they do not have a *stand-alone* code body that implements the malicious semantics. In comparison, traditional code injection attacks or persistent software parasites [5] usually require injecting a piece of code to the victim program and the injected code often manifests rich, distinct footprint that can be used to detect such code. In a ROC attack, since the malicious semantics is fulfilled by reusing existing functions in the victim binary, the attack only needs to apply a simple patch with a few writes to memory regions that correspond to *legal* variables in the original binary. These writes are indistinguishable from the existing writes in the binary. (2) Dynamically, the runtime behavior of the binary under attack complies with constraints dictated by the program semantics. The attack is mostly carried out by manipulating program states and duplicating existing function invocations. The duplicated "malicious" function invocations occur at a place where it is legal to do so. (3) Furthermore, since the attack reuses communication protocol implementation in the binary, from the network's perspective, the victim binary could still follow the communication protocol without exhibiting any anomalous behavior.

A typical scenario of launching a ROC attack is as follows: The attacker downloads the binary of a popular close-source freeware (e.g., a P2P sharing or streaming program) and then patches it with logic for malicious reuse of legitimate function(s). According to a study on how the top 100 malware programs in 2008 infect computers [3], the patched binary (i.e., the ROC trojan) could be disseminated by the attacker via a number of ways: downloaded (without user consent) from the Internet which accounts for 53% of malware infection; dropped by other malware (43%), email attachments (12%), browser iframe compromises (7%), software vulnearbilities (5%), and so forth.

ROC attacks are likely to succeed considering (1) the prevalence of "*drive-by downloads*" and the large existence of stealthy downloading malware (e.g., the trojan-downloader in Fig. 1(b)) and (2) the lack of universal binary integrity checking infrastructure for many close-source shareware programs today. Meanwhile, many close-source shareware programs are rich in functions that can be reused for malicious purposes, making them attractive targets of ROC attacks.

To illustrate the real threat of ROC attacks and study defense strategies against such attacks, we propose a systematic framework for detecting ROC vulnerabilities: Given a close-source binary, our framework will identify ROC vulnerabilities in it and further construct a ROC attack to confirm the true existence of such a vulnerability. Our framework also serves the purpose of demonstrating the feasibility (and simplicity) of ROC attacks and thus raising public awareness. The detection of ROC vulnerabilities involves two main steps:

The first step is *reuse-able feature extraction*. Given a subject binary and its output that can be used in malicious contexts (e.g., an email client and the emails it sends out), our framework will check if modular functions exist which are dedicated to producing that output. Such functions are potential targets of malicious reuse if their executions lead to very few *reversible side-effects*. For example, the email client logs emails sent in the sent-email folder – a side-effect that should be reversed for a spammer. Our framework employs dynamic binary analysis techniques to narrow down the reuse-able functions and quantify their side-effects.

The second step is *reuse-able function argument identification*. The key part of a ROC attack is the malicious setup of parameters to invoke the reuse-able feature function. We show that it is possible to identify such arguments *without* source code and symbolic information. Our framework adopts a runtime program state diff-ing approach, which involves running the subject binary twice – with the same setting but different input value assignments. The differences in the two resulting memory states will reveal a wealth of information about the arguments of the reuse-able function, including their memory regions and reference paths.

Our framework also includes a ROC attack composer. To implant malicious logic, reusable function invocations in the original binary are patched to expose critical internal states and allow mutation. Such functions and states are identified by the ROC vulnerability detector. If needed, function invocations can be duplicated in the same context of the original invocation such that the semantic constraints imposed on legal calling of the target function are satisfied, i.e., the legal calling context is maliciously reused. We provide API functions to enable easy ROC attack composition. The attacker can construct non-trivial attacks by writing a few lines of code, which will be translated into binary and patched into the victim binary.

We have implemented a prototype of the ROC vulnerability detector and attack composer and applied them to a number of real-world binaries. Our experimental results show that ROC attacks are real and easy to construct. Moreover, our framework is able to identify specific reuse-able functions and construct the corresponding attacks. For example, the email client `pine` and `mailx` can be converted into a stealthy email interceptor; the P2P software `Mutella` can be exploited to perform covert Command and Control (C&C) for a botnet (Section 5); and the P2P software `gift` can be converted to transfer sensitive files to other hosts without being noticed.

2 Approach Overview

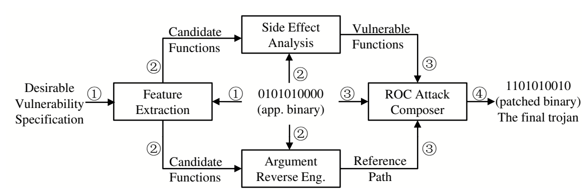


Figure 2. Typical workflow of ROC vulnerability detection and attack construction.

Fig. 2 illustrates a typical workflow of ROC vulnerability detection and attack composition. Given a target binary, the user will first specify a *desirable* ROC vulnerability. Unlike traditional "syntactic" vulnerabilities such as buffer-overflows, ROC vulnerabilities are highly dependent on the victim program's *semantics*, namely the program's functional feature that can be reused in a malicious context. The ROC vulnerability specification indicates such a desirable feature.

Using the desirable vulnerability specification as input, the *feature extraction* component automatically identifies a set of candidate functions to reuse. The best candidate function is the one that leads to the least amount of side effects. The functions' side-effects are quantified by the *side effect analysis* component. Meanwhile, the *argument reverse engineering* component identifies the memory locations of the functions' arguments. The output of this component is a *reference graph*, which presents a hierarchical view of the memory for the argument variables. Finally, using the outputs of *side-effect analysis* and *argument reverse engineering*, the *ROC attack composer* generates the actual malicious patch that invokes the best reuse-able function.

3 ROC Vulnerability Detection

3.1 Specifying ROC Vulnerabilities

Since we assume neither the source code nor in-depth understanding of a victim binary, the only thing we can leverage to define a functional feature is the input and output of the software. In many cases, the input/output does provide a lot of information of the relevant features. For instance, if we want to decide if the email sending feature of `pine` can be exploited, the email messages emitted by `pine` can be used to trace back to the functions that are responsible for sending emails. Then, the detector can further analyze these functions to see if they can be reused. For another example, if we want to detect whether the file transfer feature of a P2P client is vulnerable, we can annotate the network packets belonging to the file transfer protocol sent by the software. With the annotations, the functions corresponding to file transfer can be disclosed by execution monitoring.

To generalize the above examples, our approach to specifying ROC vulnerability is *to represent candidate features of a software by specifying the outputs generated (the inputs processed) by these features from the whole program output (input).* The specified outputs (inputs) often follow standard formats that can be inferred from high level understanding of the software. More formally, we consider the output (input) as a sequence of bytes and the relevant output (input) is a sub-sequence. The sub-sequence is described by a grammar $\mathcal{G}$. The corresponding parser filters all the irrelevant outputs (inputs). In practice, the sequence is the events recorded in the log file. Logging is done by intercepting system calls. To use our ROC vulnerability detection components, the user only needs to provide the grammar $\mathcal{G}$, which can be written according to published formats. For instance, the grammar of email messages can be easily derived from RFC-2822. The generated parser is responsible for recognizing the relevant outputs and parsing them into fields (nonterminals). As to be shown, such fields will be used to compose ROC attacks.

$$\begin{array}{rcl} Message & \rightarrow & Header\ Body \\ Header & \rightarrow & Subject\ Receiver\ Sender \\ Receiver & \rightarrow & Addr^+ \\ Sender & \rightarrow & Addr \\ Title & \rightarrow & String \\ Body & \rightarrow & String \end{array}$$

Figure 3. Simplified grammar $\mathcal{G}$ of email messages, provided as the input to the ROC vulnerability detector.

A sample output grammar provided to our detector is shown in Fig. 3. It is to detect ROC vulnerabilities in `pine` regarding the email sending feature. It is a simplified version for sake of presentation, a full grammar can be found in RFC-2822. Similarly, other grammars can be provided if the user wants to detect ROC vulnerabilities regarding different features.

3.2 Detecting ROC Vulnerabilities

This section describes how the detector works given the specification described in the previous section. For brevity, our discussion in this section focuses on *output* based specification, i.e., $\mathcal{G}$ is a grammar that filters output. Handling input relevant ROC vulnerabilities can be easily inferred and examples of input relevant ROC vulnerabilities can be found in Section 5.

3.2.1 Feature Extraction

Given a grammar $\mathcal{G}$ describing an output sub-sequence, *feature extraction* identifies the set of modular functions in the binary that are exclusively dedicated to the feature of manipulating and emitting the output described by $\mathcal{G}$. Other modular functions are less vulnerable as subverting them may cause unexpected effects. For example, the function `sendpacket` is used by a lot of features in `pine` including sending emails and communicating with email servers. The function is not vulnerable to ROC attacks regarding email sending because subverting the function introduces undesirable effects for all the services relying on the function.

Feature extraction is mainly carried out by profiling. Let o be the output sub-sequence accepted by $\mathcal{G}$ and o_i represent the ith byte of o. Our technique instruments the binary to support a mapping from an observed byte to the definition point of the byte, represented as pc_i, meaning the ith instance of instruction at pc. The instrumentation is a standard dynamic program dependency tracking (namely taint analysis), which has been widely used in data life time tracking [12], exploit detection (e.g.g, [25]), malware analysis (e.g., [29]), and so on. In particular, we instrument each memory read, write, data movement, to catch dependencies between data definition and uses. Also, we capture the call stack context of data definitions and uses.

The next step is to analyze executions to identify functions that are dedicated to producing the relevant output. Given the sub-sequence o, a standard approach would be to perform dynamic slicing [21] on o to isolate the relevant executions. Dynamic slicing is a technique proposed as a debugging aid. Given a value at an execution point, called the slicing criterion, it computes a transitive closure along program dependencies. A feature can be extracted by aggregating slices across multiple runs to find out modular functions that are dependent on by the specified outputs. However, we found that such an approach is not optimal for our purpose because it often isolates functions that do not directly manipulate the specified outputs. For example, `pine` needs to call a few initialization functions to set up the sender's environments. Such functions are dedicated to email sending and caught by slicing. However, these functions do not directly manipulate the specified outputs so that subversion through these functions is fruitless.

In our solution, given an execution E whose relevant output is o, a dynamic call tree is constructed, with a node representing a dynamic function instance and an edge $f \rightarrow g$

Content	Call Tree Paths (Calling Contexts) of Definitions
`EHLO [10.0.0.4]\r\n`	...call_mailer→smtp_open_full→smtp_ehlo→sprintf→vsprintf→vfprintf→_IO_default_xsputn
`RSET\r\n`	...call_mailer→smtp_mail→smtp_send→0x804ad38→strcpy
`MAIL FROM:<alice@bob.com>\r\n`	...call_mailer→smtp_mail→smtp_send→0x804ac58→sprintf→vsprintf→vfprintf→_IO_default_xsputn
`RCPT TO:<alice@bob.com>\r\n`	...call_mailer→smtp_mail→smtp_send→0x804ac58→sprintf→vsprintf→vfprintf→_IO_default_xsputn
`DATA\r\n`	...call_mailer→smtp_mail→rfc822_output→post_rfc822_output... →pine_header_line→0x804ac58→sprintf→...
`Date: Wed, 22 Oct 2008 14:00:...`	...call_mailer→smtp_mail→rfc822_output→post_rfc822_output... →pine_header_line→fold→sstrcpy
`From: Alice <alice@bob.com>\r\n`	...call_mailer→smtp_mail→post_rfc822_output→pine_rfc822_output→pine_rfc822_header→pine_address_line
`X-X-Sender: alice@bob.com\r\n`	...call_mailer→smtp_mail→post_rfc822_output→pine_rfc822_output→pine_rfc822_header→pine_address_line
`To: bob@alice.com\r\n`	...call_mailer→smtp_mail→post_rfc822_output→pine_rfc822_output→pine_rfc822_header→pine_address_line
`Subject: a test\r\n`	...call_mailer→smtp_mail→rfc822_output→post_rfc822_output... →pine_header_line→fold→sstrcpy
`Message-ID: <Pine.LNX....137@lo...`	...call_mailer→smtp_mail→rfc822_output→post_rfc822_output... →pine_header_line→fold→sstrcpy
`Content-Type: TEXT/... format=...`	...call_mailer→smtp_mail→post_rfc822_output→pine_rfc822_output→pine_rfc822_header→pine_address_line
`aaaaaaaaaaaaaaaaaaaaaaaaa\r\n`	...call_mailer→smtp_mail→rfc822_output→post_rfc822_output... →gf_local_nvtnl→gf_terminal→l_putc
`.\r\n`	...call_mailer→smtp_mail→smtp_send→0x804ad38→strcpy
`QUIT\r\n`	...call_mailer→smtp_close→smtp_send→0x804ad38→strcpy

Table 1. An email string and the call tree paths to function instances that define the individual string.

representing a dynamic invocation from f to g. Note that it is a tree instead of a graph as dynamically one callee instance has only one caller instance. Each byte o_i in o is then annotated on a node in the dynamic call tree if o_i is defined in the function instance represented by that node. A function instance f is said a *containing function* of o if it is the common ancestor of all the function instances annotated. Intuitively, it means the entire o is defined inside f, either directly in f or in function instances transitively invoked by f. Note that if f is a containing function, its ancestors in the dynamic call tree are also containing functions. For example, assume we want to subvert the email sending feature in `pine`. Email messages are annotated as relevant from all the outputs of `pine` according to the provided $\mathcal{G}$. Table 1 shows a sample email and the paths in the dynamic call tree that lead to function instances that define individual bytes in the email message. These paths correspond to the calling contexts of the definition points. Consecutive bytes with the same path are aggregated and shown in column `Content`. Note that the call paths are partial as they all share the same prefix `main`→`compose_mail`→`pine_send`→`call_mailer`. According to the above definitions, `call_mailer`, together with `pine_send`, `compose_mail`, etc., are containing functions.

Not all containing functions are vulnerable. We exclude functions that can be invoked in executions that do not produce the specified output. Let the set of containing functions for an execution E be $\mathcal{CF}(E)$, and the set of functions invoked by an execution E be $\mathcal{F}(E)$. Assume a test suite $\mathcal{T}$ with $\mathcal{T}^{\mathcal{G}}$ being the set of executions that manifest the relevant output. The set of feature functions is computed as follows.

$$\textit{feature}(\mathcal{G}) = \bigcap_{E \in \mathcal{T}^{\mathcal{G}}} \mathcal{CF}(E) - \bigcup_{E \in \mathcal{T} - \mathcal{T}^{\mathcal{G}}} \mathcal{F}(E)$$

That is to say, the set of feature functions include the common containing functions shared by all cases that produce relevant output, excluding those occur in any case that does not produce relevant output. In the `pine` example, `compose_mail`, `pine_send`, and `call_mailer` are the feature functions. Function `main` is not part of the feature as it occurs in executions that do not send emails.

3.2.2 Side Effect Analysis

ROC attack aims to reuse existing application logics implemented in modular functions to achieve the malicious goal. They often entail duplicating calls to feature functions in their original context. One of the necessary conditions is that the function invocation to be duplicated has to have no or very few side effects. Otherwise, benign execution will get perturbed such that stealth cannot be preserved.

Therefore, the next step of ROC vulnerability detection is to analyze the side effects of the functions in the feature we extracted in the earlier step. In this work, *a side effect of a function instance is defined as a memory write in the function instance that is used after the function instance returns or a library call that results in observable external behaviors like updates to a log file.* Writes to stack variables in the frame of a function instance f and to heap structures allocated and then freed inside f do not induce any side effects. The analysis is implemented by tracing memory writes, system calls, heap allocations and de-allocations. Details are elided.

Applying the side effect analysis to `pine`'s feature shows that all the functions in the feature do have side effects. As shown in Section 5, methods `compose_mail`, and `pine_send` have a large number of side effects. In contrast, a maximum of 18 writes to global variables and a maximum of 9 heap allocations are observed as the side effects of `call_mailer`. They can be reversed by restoring the values of the updated memory locations. Therefore, we consider `call_mailer` to be potentially vulnerable. In comparison, some side effects are not reversible like GUI displays. Functions having these side effects are not vulnerable. If none of the feature functions is vulnerable, the software is not vulnerable.

3.2.3 Reverse-Engineering Critical Arguments

After deciding feature functions and excluding functions with irreversible side effects, we have narrowed down the vulnerable functions to a small set. To decide whether they are truly vulnerable, we need to check if the behavior of these functions can be mutated by changing program state. Therefore, the last step in ROC vulnerability detection is to identify critical arguments of these feature functions. Without loss of generality, we consider one feature function f in this section.

The ROC vulnerability detector relies on checking two conditions. One is to *identify the important variables (memory regions) whose values need to be modified in order to manipulate the specified output*. For example, email re-direction entails finding the memory region that stores the recipient email address. The other condition is to *identify the reference paths to these variables (memory regions)*. A variable or a memory region cannot be simply accessed through their absolute addresses, which may change from run to run. Therefore, an attack cannot be constructed (and hence f is not vulnerable) unless a reference path that consistently leads to the same variable (memory region) across all runs can be identified.

Given one run, a simple approach to locating the memory region that stores the sensitive information is to scan the memory. However, such an approach cannot be generalized. The program may parse and then store the information to its own formats, e.g., an IP address can have multiple internal representations. Furthermore, the information may even be encrypted such as in SSL communications. In these cases we cannot simply conclude the information is not accessible and hence the program is not vulnerable.

Our ROC vulnerability detector identifies critical memory regions through memory differencing. We acquire an extra *execution* by changing some of the program inputs and directing the software to produce different outputs. The original execution is called the *reference execution*. The memory snapshots of the two executions at the invocation of the feature function f are compared to isolate the relevant memory regions. For example, in the `pine` case, the reference execution sends a message to an address x, whereas the extra execution is acquired by sending the same message to a different address y. The memory states before the invocations of `call_mailer` in the two respective runs are compared to identify the memory region that stores the recipient address, which should be the only difference of the two runs. Recall that `call_mailer` is the candidate vulnerable function detected in the earlier phase.

In practice, a dynamic data structure d may be allocated to different locations in the two runs. Comparing the memory location of d in one run to the same location in the other run may be equivalent to comparing d to a different data structure d', and hence lead to the wrong conclusion that d does not hold the same value in the two runs. In order to properly compare two memory snapshots, our detection technique needs to construct the correspondences between memory cells. We define the problem as a *memory alignment* problem. More formally, *given two executions E and E' and a memory variable i in E, the memory alignment function identifies a memory variable in E' that corresponds to i.* The function is denoted as $\mathcal{MA}_{E\rightarrow E'}(i)$, or $\mathcal{MA}(i)$ for short if the two executions are clear from the context. $\mathcal{MA}(i)$ is a partial function, for i that does not correspond to any memory variable in E', $\mathcal{MA}(i)$ is undefined, denoted as $\mathcal{MA}(i) = \perp$.

Theoretically, memory alignment is an undecidable problem. We propose an approximate solution based on *Reference Graph* (RG). Intuitively, RG identifies the reference paths to all live memory regions. Because for any live memory region, there must exist a reference path starting from a global variable, a stack variable on the current frame, or a register, and hence the roots of RG have to be one of the above three types of variables. RG serves as an indexing scheme over the memory space so that indices can be used to identify memory alignment. The formal definition of RG is presented as follows.

Definition 1 *A reference graph is a pair $\langle N, E\rangle$ with N being the set of nodes and E being the set of edges. A node represents a memory region or a field. There are two types of edges.*

- *There is a field edge between nodes n and m, denoted as $n \multimap m$, if m is a field of n. The field name is annotated on the edge. If symbolic information is not available, the offset is annotated.*
- *There is a pointer edge between nodes n and m, denoted as $n \longrightarrow m$, if n stores a pointer that points to m.*

In our `pine` example, we acquire two executions by running `pine` twice, with the same configuration and the same sender and recipient addresses, but different subjects and email contents. We show these two test emails in Table 2: one is a spam email and the other is a regular one.

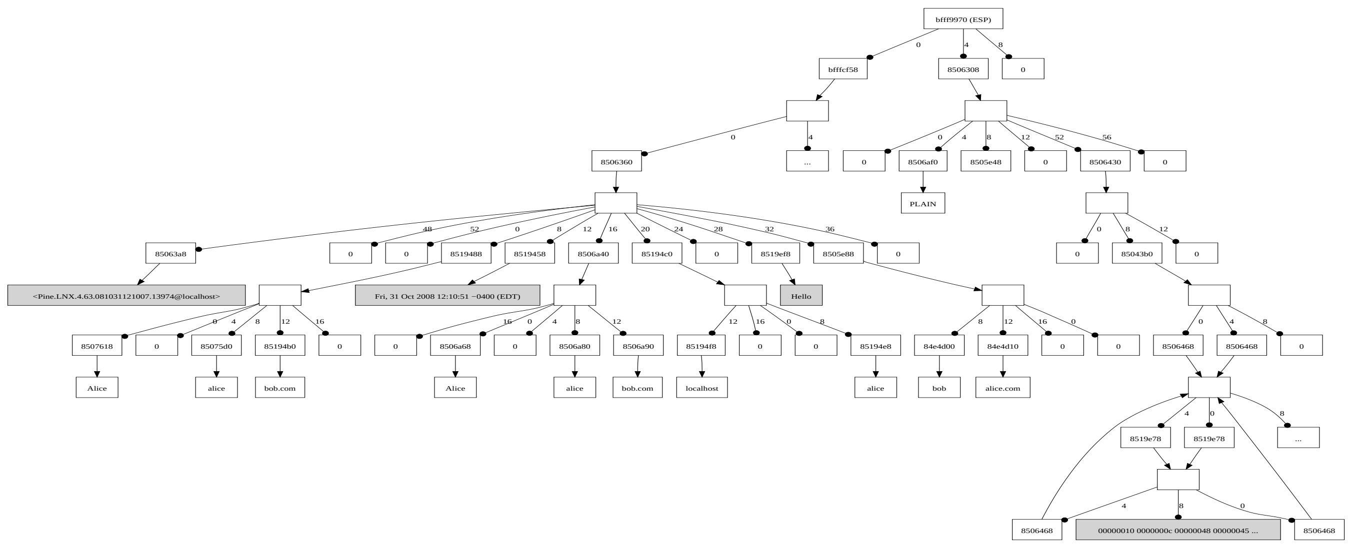
`Subject: SPAM`	`Subject: Hello`
`From: <alice@bob.com>`	`From: <alice@bob.com>`
`To: bob@alice.com`	`To: bob@alice.com`
`This is a spam email.`	`Hello, world`

Table 2. The two different test emails

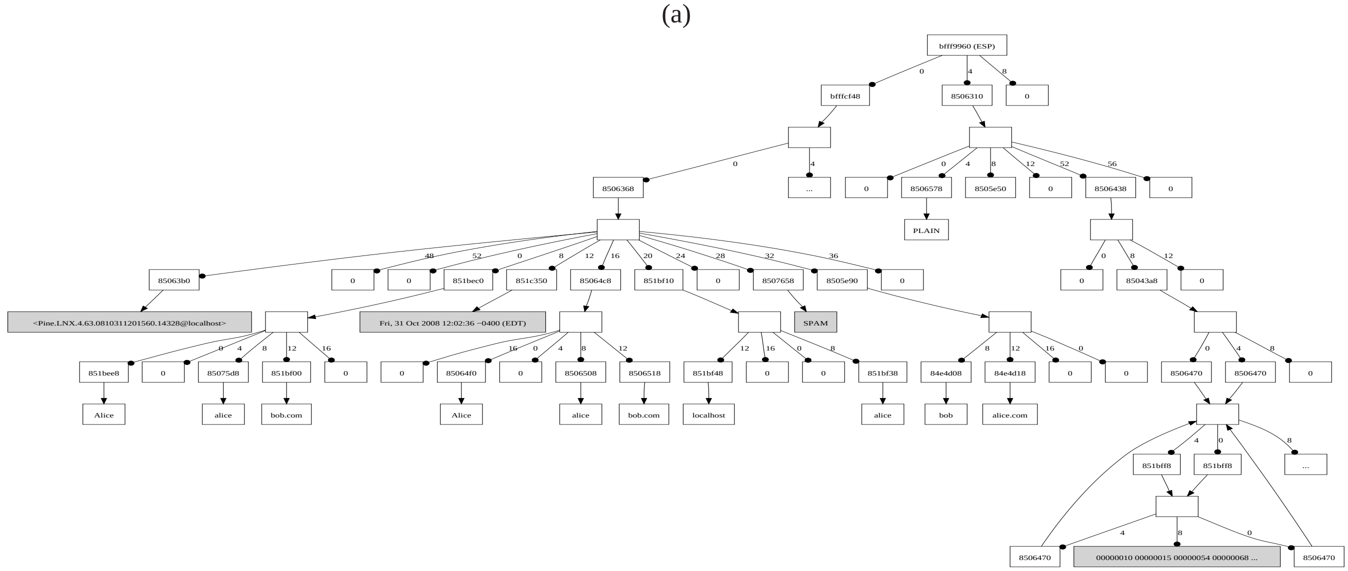
The two RGs at the invocation point of `call_mailer` are presented in Fig. 4. The root nodes represent the current stack frame (the roots for the global regions are irrelevant for our discussion and thus omitted). In Fig. 4(a), three fields have been reverse engineered with the byte offsets of 0, 4 and 8. The first two are pointers, the last one contains a value 0. The first pointer field `0xbfffcf58` points to a memory region that has two fields, and so on.

The two memory snapshots are aligned by aligning their RGs. Since RGs are graphs with labels, their alignment can be carried out by a simple labeled graph alignment algorithm, which will not be further discussed due to the space limit. A memory difference is defined as a memory region that has a different value in its alignment in the other RG. Observe the two RGs in Fig. 4 are highly similar. The differences are highlighted in the figure. Note that pointer value differences are ignored to tolerate non-determinism in memory allocation. Two out of the four differences are for the subject and the content. The other two are for different time-stamps and bookkeeping information. Note that the content is encoded, which justifies our approach of memory diff-ing because a simple scan over the memory would fail to find the content.

Besides identifying critical memory regions, the other goal of RG is to provide reference paths to these regions. A reference path is a RG path that starts from a root and leads to the destination region. It represents how to address the region at the current execution point. The software is vulnerable only if such paths can be reverse engineered, because then a ROC attack can be easily composed by mutating the values of these regions. In Fig. 4, the reference paths from the roots to the differences can be discovered from the RGs. For example, the reference paths to the subject and the content are *(*(*(ESP+0)+0)+28) and *(*(*(*(*(ESP+4)+52)+8)+0)+0)+8), respectively. Note that dictated by the definition of memory alignment, the paths to the corresponding memory regions are identical in the two graphs, e.g. the paths to the email subject are the same. We point out that the normal



(a)



(b)

Figure 4. RGs at the invocation of `call_mailer` for (a) sending a spam and (b) sending a regular email.

execution can be mutated to the malicious one if the values in the shaded regions in (a) are copied over to the regions in (b) at the execution point where the snapshot is taken.

Reference Graph Construction. RG plays an important role in ROC vulnerability detection. Next we present an algorithm for RG construction. The pseudo-code is presented in Algorithm 1. The algorithm takes a memory snapshot S at a particular execution point, a hashmap HR that records the memory regions allocated during execution, and a hashmap HF that records the memory addresses that have been accessed. It then generates the RG at the execution point. The hash map HR is created by tracing memory allocation/deallocation functions and function entries (for stack frames), e.g., a new region is inserted when a piece of memory is allocated with the key being the base address. The hash map HF is acquired by tracing memory accesses. Any location that has been accessed has an entry in HF.

At line 2, the root nodes of the RG are the region for global variables and the region for the current stack frame. Before RG construction, registers are pushed to the stack so that they become part of the current stack frame and we do not need to create a separate root node for registers. Note that individual global variables and stack variables on the current frame become the fields of the root nodes; other stack frames can be reached from the current frame. The basic idea of the algorithm is to start from the root nodes and gradually explore all the reachable memory regions and their fields, by using a worklist. Observe that all live variables are reachable from the root nodes. The loop between lines 6 and 22 explores a region from the worklist. It traverses each offset in the region. It tests if the location denoted by the offset has been accessed ever since the region was created at line 8. If so, the offset must represent a field. A value-based heuristic is used to decide if the value stored at the current offset, denoted by $*(p)$ at line 12, is a pointer. If so, the algorithm further tests if it points to the middle of an existing region at line 13. If this

Algorithm 1 Reference Graph Construction.

Input: HR is the hashmap for regions; HF is the hashmap for memory locations occurred in any accesses; S is the snapshot.

identify the current frame fm and the global region g from S
insert fm and g to the RG.
$wl \leftarrow \{fm, g\}$
while wl is not empty **do**
  $r \leftarrow wl.pop()$
  for each possible offset o in region r **do**
    $p \leftarrow r.base + o$
    if $HF.contains(p)$ and $HF.get(p)$ happens after r **then**
      a new field f is created.
      a field edge $r \overset{o}{\multimap} f$ is inserted.
    end if
    if $isPointer(*(p))$ **then**
      if $*(p)$ points to the middle of a region $rx \in HR$ **then**
        separate rx to two regions with one starting at $*(p)$
      end if
      if $HR.contains(*(p))$ and $*(p)$ is not a region in the RG **then**
        A new region node new_r is created for $HR.get(*(p))$
        a pointer edge $f \rightarrow new\_r$ is inserted
        $wl \leftarrow wl \cup new\_r$
      end if
    end if
  end for
end while

is the case, the existing region is divided into two regions. It then tests if the pointer points to the beginning of a region, if this is true and a node has not been created for the region, a new node is created in the RG; a pointer edge is inserted; the new node is added to the worklist for later exploration. An important property of RG is that *any memory region that is reachable in the ideal reference graph, i.e., the one created with the knowledge of data structure, is reachable in the RG produced by our algorithm*. The proof is omitted.

4 ROC Attack/Trojan Composition

```
<pine_send>:
 ...
 81d178f: 89 44 24 08          mov    %eax,0x8(%esp)
 81d1793: 89 54 24 04          mov    %edx,0x4(%esp)
 81d1797: 8d 85 c4 fe ff ff    lea    0xfffffec4(%ebp),%eax
 81d179d: 89 04 24             mov    %eax,(%esp)
 81d17a0: e8 ca 43 00 00       call   81d5b6f <call_mailer>
 81d17a5: 85 c0                test   %eax,%eax
 81d17a7: 7e 0c                jle    81d17b5
 81d17a9: c7 85 18 ca ff ff 01 movl   $0x1,0xffffca18(%ebp)
```

(a) Original entry code for `call_mailer`

```
<pine_send>:
 ...
 81d178f: 89 44 24 08          mov    %eax,0x8(%esp)
 81d1793: 89 54 24 04          mov    %edx,0x4(%esp)
 81d1797: e9 xx xx xx xx       jmp    <entry_patch>
 81d179c: 90                   nop
 81d179d: 90                   nop
 81d179e: 90                   nop
 81d179f: 90                   nop
 81d17a0: e8 ca 43 00 00       call   81d5b6f <call_mailer>
```

(b) Rewrite first 9 bytes before `call_mailer`

```
<entry_patch>:
  0: 8d 85 c4 fe ff ff         lea    0xfffffec4(%ebp),%eax
  6: 89 04 24                  mov    %eax,(%esp)
  9: 60                        pushal
  a: e8 xx xx xx xx            call   <malicious_logic>
 10: e8 xx xx xx xx            call   <exit_patch>
 15: 61                        popal
 16: e9 xx xx xx xx            jmp    <81d179c>
```

(c) `entry_patch`

```
<malicious_logic>:
(1) Keep a copy of all side
effect variable
(2) prepare patched arg for
call_mailer
(3) invoke it to send SPAM
or redirect email
```

(d) Malicious logic

```
<exit_patch>:
Patch back side effect
variable
```

(e) `exit_patch`

Figure 5. The patched code that sends a copy to a malicious address.

Given a grammar specification, our ROC vulnerability detector reports feature functions and critical arguments with their reference paths. If both can be identified, the software is highly susceptible to ROC attacks. In order to decide if these candidates are true positives, we further develop an attack composer that allows user to easily construct ROC attacks.

Recall that feature functions are those that emit the specified outputs and their invocations can be duplicated for subversion if needed as they do not have irreversible side effects. Furthermore, critical arguments of these functions and their reference paths also allow mutating the arguments. Therefore, we propose a programming interface that facilitates easy ROC attack composition. As shown in Table 3, the interface provides macros that allow inserting code before or after a function invocation, or right at the beginning of the invoked function. It also supports simple argument manipulations and function call duplication. A ROC attack can be written using a C-like language with the APIs. The following code snippet illustrates a ROC attack that re-directs an email message.

Macro/Method	Description
BEFORE(int `func`) {code}	insert the code block before `func`
AFTER(int `func`) {code}	insert the code block after `func`
ENTRY(int `func`) {code}	insert right inside `func`
void get(int* `field`)	retrieve the argument `field`
void set(int* `field`, void* `val`)	set the argument with `val`
void duplicate(int `func`)	duplicate the invocation of `func`

Table 3. ROC Attack Composition API.

```
BEFORE(call_mailer){
    set(&receiver, "ghost@somewhere.com");
    duplicate(call_mailer);
}
```

The attack duplicates the `call_mailer` invocation and mutates the `receiver` of the email address before the duplicated call. The attack code is inserted before the original invocation to `call_mailer`. Note that our tool identifies the address for the given `call_mailer` function and the reference path for `receiver`. The result is that a copy of the email is sent to the malicious address before it is sent to the right receiver. The snippet is translated into assembly code, which is further compiled to a piece of independent binary. The binary is then patched to the original software. The patch is comprised of three parts: an *entry patch* that precedes the duplicate and intercepts the control flow right before the original benign invocation, a *malicious logic* that implements the main body of the attack, and an *exit patch* that reverses the side effects. The malicious logic includes accessing and changing the critical argument denoted by the field name *receiver* and making a duplicated call. The field represents the argument that decides the output value parsed by the non-terminal *Receiver* in the grammar $\mathcal{G}$, denoting the receiver's address.

Binary Patching. Without recompiling, patching is done by replacing a few instructions before the invocation sites specified in the attack code. No significant code mutation is needed. We illustrate binary patching using the ROC attack to `pine` described earlier. Fig. 5 (a) shows the original assembly code around `call_mailer`. To patch the software, as shown in Fig. 5(b), a few instructions before the invocation is replaced with a jump, which jumps to the entry patch. The entry patch first restores the replaced instructions at the call site to preserve the original semantics of the program, and then it keeps a copy of all regular registers, and makes calls to the malicious logic function and then the exit patch as shown in Fig. 5(c). At the end of the entry patch, the control flow returns to the original invocation.

5 Evaluation

We have implemented the ROC vulnerability detector using Valgrind-3.2.3 [24]. We instrument binary to (1) collect memory reads, writes, data dependencies, heap allocations, and de-allocations, along with the call stack contexts; (2) keep track of function live ranges, caller-callee relations; and (3) take snapshots of memory along with regular registers for reference graph construction at selected function invocation points. Feature extraction, side effect analysis, and reference graph based memory comparison are conducted off-line based on the trace file.

The ROC attack composer is implemented independently. We design a C-like script language. A program written in this language can be translated to assembly code and further compiled to binary. The attack binary is then integrated into the original binary. Variables declared and used exclusively in the attack code (e.g., a loop index) are allocated at the end of a data section of the original binary. Constants such as strings are embedded into the attack code as they are immutable. More specifically, they are embedded right after function invocation instructions in the attack code. The target address of the `ret` instruction in the invoked function needs to be adjusted accordingly. The addresses of the constants can be easily computed from the program counter of the invocation instructions. In order to merge the attack binary into the original binary, besides the binary patching technique presented in the previous section, the main body of the attack code is stored in the unused space in the code segment, which can be identified from the ELF header.

We have applied our framework to a number of real-world binaries. Next we present the outcome from our ROC vulnerability detector and attack composer.

The first step in ROC vulnerability detection is to specify the grammar. Here, we assume some high level prior knowledge about the functionalities of the application such as the protocol being used. In particular, our evaluation mainly involves two protocols, an email protocol (RFC-2822) and a P2P protocol Gnutella-0.6. We aim to detect ROC vulnerabilities in the various implementations of these protocols. We take 5 widely used software as the benchmarks which are shown in details in Table 4 and Table 5. The `Size` in Table 4 is the binary size. In the email implementations (`pine` and `mailx`), we aim to find the feature which is responsible for email sending so that we can use to redirect email or send spam. In the P2P implementations (`mutella`, `peercast`, and `gift`), we aim to implant malicious logic such as a C&C channel.

Table 4 shows the cost of profiling in the feature extraction phase. The profiling consists of one expensive instruction level profiling and 10 times featherweight function level profiling. The instruction level profiling collects memory reads, writes and dependencies and produces large log files. It is to facilitate identifying containing functions. The function level profiling is to identify containing functions that are not dedicated to the feature, i.e., containing functions executed in runs that do not produce the specified output (or do not accept the specified input). The overall cost is presented in Table 4. The overall profile time, the maximal number of traced threads for one run, and the total log size are shown in the 3^{rd}, 4^{th}, and 5^{th} columns, respectively. Note that `libGnutella` is a plug-in in `gift`. They are treated as two different benchmarks because we are interested in their different features, namely, the file index management feature in `gift` and the file transfer feature in `libGnutella`. The first instruction level profiling is the dominant factor in the cost. Currently, it collects traces for the entire execution which is sub-optimal. We will work on optimizing this component in the future.

Benchmark		Time	#Traced Threads	Log Size
Software Name	Size			
pine-4.63	6.3M	8m25s	1	6.4G
mailx-12.4	712K	5m48s	1	2.9G
mutella-0.4.5	843K	10m16s	9	8.2G
peercast-0.1217	58K	15m18s	5	3.5G
gift-0.11.8.1	321K	7m57s	1	2.2G
libGnutella.so.0.11	657K	12m36s	1	3.1G

Table 4. Cost of profiling in feature extraction.

Table 5 summarizes the input and outcome of the detector. Columns in `Prior Knowledge` presents the information provided by the user: `Protocol` is the feature represented by the provided grammar. Column #`Var` shows the number of critical arguments, which correspond to some non-terminals in the grammar. Columns in `Observed Feature Functions` show the extracted feature functions. Note our techniques do not require any symbolic information, and we present function name mainly for readability. The next three columns show the maximal length of the reference paths of the critical arguments, the number of critical variables that are identified, and the number of containing functions. The side effect columns present the number of writes to global variables (`#G`), heap variables that are live at the end of the function (`#H`), and external files (`#F`). The performance overhead column shows the runtime overhead of memory tracing and comparison. Note that in this phase, we do not need to emit external traces as the demanded hash maps are maintained on the fly and used at the end, and also we do not need to capture data dependency as it is done in the previous phase. The slowdown factor is acquired by comparing with the time of running the program on Valgrind without any instrumentation. We did not collect the overhead data for daemon programs as they are event driven and do not execute continuously. From the collected data, the overhead factors are quite stable at roughly 1.7X. If feature functions and all the specified critical arguments can be identified, we consider the software vulnerable regarding the specified feature.

In order to identify false positives, we use our attack composer to construct ROC trojans. If an attack can be constructed, the reported vulnerability is confirmed. Table 6 summarizes the attacks. Due to space limitation, we next present three representative cases in detail.

Attack Description	Benchmark	Patch Binary Size (bytes)	Succeed?
Email Redirection	pine-4.63	486	✓
	mailx-12.4	320	✓
Email Spamming	pine-4.63	1192	✓
	mailx-12.4	-	×
Covert C&C	mutella-0.4.5	1460	✓
File Transferring	gift-0.11.8.1 libGnutella.so.0.11	234 670	✓

Table 6. Summarized result from the ROC attack composer.

Benchmark	Prior Knowledge		Observed Feature Function		Max Length of Ref Path	#Identified Var	#Containing Functions	Side Effect Write			Performance Overhead
	Protocol	#Var	Func Addr	Func Name				#G	#H	#F	
pine-4.63	RFC-2822 Email Sending	4	0x081c613c	compose_mail	1	1	7	183	9	1	1.71X
			0x081cbf67	pine_send	3	0	8	181	37	1	1.68X
			0x081d5b6f	call_mailer	6	4	9	18	9	0	1.72X
mailx-12.4	RFC-2822 Email Sending	4	0x08090f59	talk_smtp	3	3	10	3	2	0	1.77X
			0x08092306	smtp_mta	3	3	9	9	1	0	1.77X
			0x0808e864	start_mta	3	3	8	18	1	0	1.64X
			0x0808e6a2	transfer	3	3	7	18	1	0	1.61X
			0x0808ee02	mail1	3	3	6	70	1	2	1.60X
mutella-0.4.5	Ping Send	1	0x080d0cc2	MGnuNode::SendPacket	5	1	15	1	1	0	-
			0x080d2eb8	MGnuNode::Send_Ping	4	1	14	1	1	0	-
	Ping Recv	1	0x080d64e2	MGnuNode::HandlePacket	5	1	8	-	-	-	-
			0x080d1b1c	MGnuNode::Receive_Ping	4	1	9	-	-	-	-
peercast-0.1217	Ping Send	1	0xb7eee13e	GnuStream::ping	1	1	9	0	6	0	-
			0xb7eedf5a	GnuStream::sendPacket	3	1	8	0	6	0	-
	Ping Recv	1	0xb7eef3b6	GnuStream::processPacket	6	1	8	-	-	-	-
gift-0.11.8.1	Index Management	0	0x08054923	share_update_index	5	0	16	-	-	-	-
			0x0805489e	update_index	5	0	17	-	-	-	-
libGnutella.so.0.11	Query Recv	1	0xb7dc522a	recv_packet	3	2	21	-	-	-	-
			0xb7d027fe	gt_msg_query	3	1	22	-	-	-	-
	Ping Recv	1	0xb7d01659	gt_msg_ping	4	1	22	-	-	-	-

Table 5. Summarized result from the ROC vulnerability detector.

(1) Pine. We are interested in subverting the email sending feature of `pine`. The grammar was presented in Section 3. Four critical arguments are specified, namely, `sender`, `receiver`, `subject`, and `content`. Three feature functions are identified. All the 4 critical arguments are disclosed at `call_mailer` while only 1 and 0 are identified at `compose_mail` and `pine_send`, respectively. More importantly, these two functions have a much larger number of side effects with irreversible file side effects. Therefore, `call_mailer` is highly vulnerable and thus `pine` is vulnerable. We have constructed an email re-direction attack in Section 4. The patched binary is 486 bytes. Observe that this extra code is small compared to the functionality realized, attributed to its reuse oriented composition. The attack is stealthy as the original email is sent to the original receiver without any signs of being duplicated. The extra message sent is not recorded in the log. There is no observable change on the user display. `Pine` can also be easily turned into a spam sender by changing the subject and content of the email and then duplicating the invocation of `call_mailer`. The extra code has 1192 bytes.

(2) Mailx. The case of `mailx` is very similar to `pine`. It is also vulnerable regarding email sending. The difference lies in that 5 feature functions are identified and 4 out of 5 are almost equally vulnerable (`mail1` is not vulnerable due to the file level side effect). Furthermore, one critical argument `content` can not be reverse engineered for all these functions so that `mailx` can not be mutated to a spam sender by our technique. Inspecting the source code shows that a temporary file is used to store the email body so that it is not present in the memory. Nonetheless, the redirection attack can be successfully constructed with a piece of 320-byte binary code being added to the original binary.

(3) Mutella. *Malicious intent and desirable features:* In this case, we are interested in stealthily introducing a covert Botnet command and control (C&C) mechanism to the `mutella` implementation. The idea is to reuse the Gnutella (the protocol used by `mutella`) internal management protocol such that network packets would look normal and the C&C overlay is completely invisible on the peers. In particular, from the Gnutella protocol specification [2], we know a "PING" packet is used to announce the presence of a node on the network, and other peers respond with a "PONG" packet to notify they are reachable. The "PING" message is also forwarded to other connected peers if the hops are still alive. We can encode various botnet commands by sending the identical "PING" packet in a sequence with various lengths. Note that doing so is completely legal according to the protocol specification (as such behavior corresponds to a node keeps trying to find its neighbors). Un-infected peers would work normally with infected peers and only infected peers understand these encodings among themselves.

Reuse-able function identification: Therefore, we provide the PING message grammar to the ROC vulnerability detector with the critical argument being GUID (the identification of a message). Note that we are interested in both the sending and receiving PING message features. They are considered as separate features as they are implemented by different sets of functions. For both the PING send and the PING receive features, two feature functions and the critical argument are identified, indicating that `mutella` is vulnerable.

We select `Send_Ping` and `Receive_Ping` to compose the attack. Part of the attack code is presented as follows.

```
BEFORE(Send_Ping) {
    for(i=0;i<2;i++){//Command A
        duplicate(Send_Ping);
    }
}
...
ENTRY(Receive_Ping) {
    get(&GUID);
    if(two consecutive messages with identical GUID)
        do_command_A();
}
```

Attack logic composition: The patch duplicates the invocation of `Send_Ping` and wraps the duplication into a loop, which iterates a number of times depending on the command that we want to deliver to other peers. To complete the C&C channel, the lower half of the attack code handles the receiving end of the "PING" messages to decode commands. It gets the argument GUID at the invocation to Receive_ Ping and decodes the command based on the number of consecutive messages with the same id and takes the corresponding action. The get() function concerns input instead of output. It is translated to a memory access following the reference path to the reverse engineered argument GUID, which is *(ESP+0) in this case. Moreover, as feature functions concerning input most likely do not get duplicated, our detector does not analyze their side effects, which explains the '-' symbols in the side effect columns. Overall, the patch adds 1460 bytes.

We performed a small scale deployment of the patched mutella. Two commands were implemented to instruct an infected peer to print two different messages. One peer served as the bot-master, whose patch on the sending side, i.e., the patch at the invocation of Send_Ping, regularly reads an external file, which contains the command. If a command is specified, it then propagates this command through the covert C&C channel to instruct its peers to print the message. Otherwise the patched mutella runs completely normally.

Other experiments. The case of peercast is very similar to mutella. Our detector flags it as vulnerable regarding the send "PING" and receive "PING" features. For the case of gift and libGnutella (the latter being a plug-in in the former), we reused their index management feature as well as the messages "PING" and "Query" to implant a secret *file transfer channel*. We leave details of this experiment in our technical report [22] due to lack of space.

6 Discussion

Having demonstrated the feasibility of ROC attacks and their potential threats, we now discuss possible approaches to ROC attack detection and prevention.

Binary integrity check. The most intuitive way to detect ROC attacks is to hash all legal binaries (e.g., using Tripwire [20]) and periodically check their integrity. In practice, however, it is difficult to maintain up-to-date, globally consistent hash values, considering the frequent, automatic software patching and update, as well as the decentralized distribution of binaries and patches. Moreover, users may not always enforce timely binary integrity check [1]. This, in part, explains the prevalence of trojans and other drive-by downloads on today's Internet. Meanwhile, it is also impossible to hash all malware (including trojans) samples for their detection, due to the large amount and the dynamics of today's malware [11].

Control flow integrity check. A ROC attack does not violate control flow integrity except at the entry and exit points where the malicious patch gets the control. Therefore it may be possible to detect such violations by monitoring and profiling the binary's normal control flows and enforcing them at runtime. For example, we could use CFI [6] to enforce legal control flow transfers at those entry/exit points. One challenge would be that, since the CFI enforcement itself is *part of* the victim binary, the ROC attacker may bypass the CFI check as part of its side-effect elimination patch.

Host-based IDS. ROC attacks are carried out by duplicating existing, legal function invocations. As such, the attacks may be oblivious to many host-based intrusion detection systems (e.g., VtPath [15]). However, the timing/sequencing characteristics of the duplicated feature function invocations may provide a lead for their detection. Hence, detectors based on behavioral sequence analysis (e.g., [18, 16]) may be able to detect ROC attacks.

Network-based IDS. ROC attacks are able to preserve the normal network behavior of the victim binary, as demonstrated by the mutella case study. As such, most network-based IDSes (e.g., PAYL [27]) would not pickup behavior abnormality. However, depending on the nature of a specific ROC trojan, it is possible that an NIDS using content-based signatures be able to detect its malicious traffic (e.g., spams). Such detection, unfortunately, cannot be generalized to all ROC trojans.

To prevent ROC attacks, one way is to break the software modularity, e.g., by transforming a program so that it contains very few function calls, which can no longer be singled out to perform a malicious action without few side-effects. Another approach is to obfuscate the binaries so that it would be difficult to identify reuse-able functions. In fact, many malware programs in the wild adopt such strategy to avoid detection. We argue that legitimate programs may also benefit from obfuscation in preventing ROC attacks.

7 Related Work

Return-into-libc attack. The ROC attack is related to the return-into-libc attack [14, 23]. The return-into-libc attack requires prior knowledge about the implementation of the returned library functions and is defeat-able by randomization techniques [19]. On the other hand, the ROC attack uses dynamic program analysis techniques to infer the reuse-ability of application level functions. More importantly, the control flow deviation caused by return-into-libc attacks is fairly obvious and easily detectable; whereas ROC attacks by design try to mimic the control flow of the victim program and reverse any side-effects.

Return-oriented programming. Shacham et al. recently proposed a return-oriented programming paradigm [26, 9], which reuses existing instruction sequences in large code segments (e.g., library) to compose malicious logics. This paradigm enables reuse of very basic functionalities at the granularity of short instruction sequences; whereas ROC attacks reuse high-level functional features of software at the (much coarser) granularity of modular functions.

Feature extraction. Prior work exists in feature extractions from binaries. In software maintenance, Wong et. al. proposed an execution slice-based technique to identify the basic blocks which are used to implement a program feature [28]. Greevy et. al. proposed a compact feature-driven approach based on dynamic analysis to characterize features and computational units of an application [17]. More recently, Caballero et al. [10] independently proposed a binary code extraction technique by combining dynamic and static analysis, to extract the malware encryption and decryption functions and reuse them in a network proxy (to decrypt the encrypted traffic). They mainly focus on how to extract the transformation function in which the entry point needs to be given, inside the binary, for

[1] In fact, one purpose of this work is to promote such practice.

the purpose of malware analysis. In addition, they reuse the transformation code in a *different* program (i.e., the malware analysis program); whereas we reuse the code within the *same* binary, with additional requirements (e.g., side-effect minimization and reversal).

Program understanding. There are also a variety of methods for profiling, testing, slicing, and debugging program behavior for a given binary. In particular, data structures reveal a wealth of information for program understanding. Recent efforts have applied machine learning techniques to infer the data structures of a binary from a memory snapshot [13]. Our experience shows that such data structure inference techniques are not accurate enough for reference graph construction in generating the patches for ROC attacks.

Memory Graph. Our reference graph (RG) concept is similar to the object reference graph for garbage collection in object oriented programs [7] or the memory graph [30] in C programs. An object reference graph has objects as its nodes connected through their field edges. It mainly focuses on the management of dynamically allocated memory. A memory graph has dynamic data structures as its nodes and "points-to" relations as its edges. Memory graphs require prior knowledge about data structure definitions [30]; whereas our technique for ROC attack construction assumes only binaries. In addition, the requirement of RG is less stringent, meaning that an RG is valid as long as it provides valid reference paths to specific memory regions without requiring the nodes and edges to precisely follow the actual data structure definition. The garbage collector by Boehm [8] also traverses memory to find reachable regions without demanding symbolic information. It does not explicitly build the reference graph and its traversal is coarse-grained, without capturing field information.

8 Conclusion

The ROC attack/trojan poses a new threat, virtually transforming a legal binary into a stealthy, malicious one. The neutral functional features in a legal binary are potential targets of ROC attacks. ROC trojans are heavily dependent on the semantics of their victim binary programs and there exists no generic content or behavior "signature" across all ROC attacks. To defend against ROC attacks, we present an integrated framework for the detection of ROC vulnerabilities in a binary and for the construction of concrete ROC trojans. Our experiments with a number of real-world software binaries indicate that the ROC attacks are real and ROC vulnerabilities can be detected and confirmed in a systematic fashion.

References

[1] Bitdefender malware and spam survey. *http://news.bitdefender.com/NW1094-en–BitDefender-Malware-and-Spam-Survey-finds-E-Threats-Adapting-to-Online-Behavioral-Trends.html*.

[2] Gnutella protocol spec. *http://wiki.limewire.org/index.php?title=GDF*.

[3] Most abused infection vector. *http://blog.trendmicro.com/most-abused-infection-vector/*.

[4] Virus collection (vx heavens). *http://vx.netlux.org/vl.php*.

[5] Parasitic malware: The resurgence of an old threat. *Network Security*, 2008(3):15 – 18, 2008.

[6] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM CCS*, 2005.

[7] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. *SIGPLAN Not.*, 33(5):269–279, 1998.

[8] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM PLDI'93*, 1993.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proc. of the 15th ACM CCS*, 2008.

[10] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proc. of the 17th NDSS*, 2010.

[11] J. Canto, M. Dacier, E. Kirda, and C. Leita. Large scale malware collection : lessons learned. In *Proc. of the 27th IEEE SRDS*, 2008.

[12] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[13] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceeding of 8th Symposium on OSDI*, 2008.

[14] S. Designer. "return-to-libc" attack. *Bugtraq*, August 1997.

[15] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proc. of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[16] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proc. of the 12th NDSS*, 2004.

[17] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proc. of the Ninth European Conference on Software Maintenance and Reengineering*, 2005.

[18] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Computer Security*, 6(3):151–180, 1998.

[19] A. D. Keromytis. Randomized instruction sets and runtime environments past research and future directions. *IEEE Security and Privacy*, 7(1):18–25, 2009.

[20] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proc. of the 2nd ACM CCS*, 1994.

[21] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[22] Z. Lin, X. Zhang, and D. Xu. Reuse-Oriented Camouflaging Attack: Vulnerability Detection and Attack Construction. Technical report, CERIAS TR 2009-29, Purdue University, 2009.

[23] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 10(58), 2001.

[24] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM PLDI'07*, 2007.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 13th NDSS*, 2005.

[26] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls. In *Proc. of the 14th ACM CCS*, 2007.

[27] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. of the 6th RAID*, 2004.

[28] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[29] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM CCS*, 2007.

[30] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, 2002.