PeerPress: Utilizing Enemies' P2P Strength against Them

Zhaoyan Xu
SUCCESS Lab
Texas A&M University
z0x0427@cse.tamu.edu

Lingfeng Chen
SUCCESS Lab
Texas A&M University
lingfeng@cse.tamu.edu

Guofei Gu
SUCCESS Lab
Texas A&M University
guofei@cse.tamu.edu

Christopher Kruegel
Dept. of Computer Science
UC Santa Barbara
chris@cs.ucsb.edu

ABSTRACT

We propose a new, active scheme for fast and reliable detection of P2P malware by exploiting the enemies' strength against them. Our new scheme works in two phases: host-level dynamic binary analysis to automatically extract built-in remotely-accessible/controllable mechanisms (referred to as Malware Control Birthmarks or MCB) in P2P malware, followed by network-level informed probing for detection. Our new design demonstrates a novel combination of the strengths from host-based and network-based approaches. Compared with existing detection solutions, it is fast, reliable, and scalable in its detection scope. Furthermore, it can be applicable to more than just P2P malware, more broadly any malware that opens a service port for network communications (e.g., many Trojans/backdoors). We develop a prototype system, PeerPress, and evaluate it on many representative real-world P2P malware (including Storm, Conficker, and more recent Sality). The results show that it can effectively detect the existence of malware when MCBs are extracted, and the detection occurs in an early stage during which other tools (e.g., BotHunter) typically do not have sufficient information to detect. We further discuss its limitations and implications, and we believe it is a great *complement* to existing passive detection solutions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

P2P, Malware analysis, Malware detection

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

Malicious software (malware) is a serious threat to Internet security. While many early botnets use centralized C&C architecture, botmasters have realized its limitations and begun to use more advanced and robust peer-to-peer (P2P) architectures for C&C [27]. For example, several contemporary successful botnets such as Storm/Peacomm and Conficker have infected millions of computers and adopted P2P techniques in their C&C coordination [2, 48]. As stated in a recent report [26], the Kaspersky Security Network detected more than 2.5 million P2P malware incidents per month in March 2010, a high water mark reached for the first time in its monitoring history. A recent P2P botnet, Sality, is still alive as of the writing of this paper and becoming more complex [12]. In short, P2P malware is widely believed to be a promising direction for future malware [27, 43, 50].

Unfortunately, to date, there is relatively little research available on detecting this important threat. Network-level detection techniques have been proposed to perform clustering/correlation analysis to identify suspicious botnet infection/behavior patterns [29, 31, 32, 41] or to analyze the network traffic graph/structure to detect possible P2P botnets [33, 34, 43]. However, suspicious pattern identification may fail in front of traffic encryption, traffic randomization and timing pattern manipulation [49]. Structure/graph analysis can only detect P2P structure regardless whether the traffic is actually malicious or not, and it typically requires tremendous resources (e.g., global ISP-level view) for acceptable results (a conclusion also mentioned in [34]), making it a less attractive solution to Enterprise networks. In another direction, host-based detection techniques such as traditional signature-based approaches (e.g., anti-virus tools) and more recent behavior-based approaches (e.g., [35, 36]) have also been proposed. However, due to the widely used advanced obfuscation/polymorphism [46] and the requirement of client-side installation, the solutions are not attractive for large scale P2P malware detection. Finally, it is worth noting that both host-based techniques and the above-mentioned network-based approaches have one common limitation because of their passive monitoring mechanism: they tend to be *slow* in terms of detection, e.g., they need to wait until some (or many) actual (suspicious/malicious) activities/communications occur to be able to detect the malware existence.

In this paper, we focus on answering the following question: *is it possible to combine both the robustness of host-*

based approaches and the efficiency of network-based approaches to provide fast, reliable, and scalable detection of P2P malware? We believe that while P2P provides more flexible and robust coordination for the enemy, we can utilize the enemy's strength against him. A key insight is that P2P malware has to have built-in remotely-accessible/controllable mechanisms. That is, P2P malware has to open some port(s) for peer-to-peer communication, which is required for providing binary downloading services to new infected machines (i.e., egg downloading [31]), or for easier later access/control by remote attackers. If we can determine the port number(s) in use and further know the access/control conversation logic through that port (we refer to this information as Malware Control Birthmarks, or MCBs, as defined in detail later), we could uniquely identify that P2P malware.

Our key insight motivates us to design a novel two-phase detection framework: (i) first, we automatically extract MCB through *host-level dynamic malware analysis*; (ii) then, with the MCB information, we perform *network-level, active, informed probing* to identify infected machines. Thus, a P2P malware sample will expose itself if it opens specific port(s) or/and it responds in a predicted way to a specific probing packet. It is worth noting that our new detection scheme applies in general to any malware that has MCBs, not just to P2P malware. For example, Trojan/backdoors also belong to the detection scope of this scheme and they are among the current most popular malware in the wild as shown in a recent Symantec Internet security threat report [8].

Our new design naturally bridges host-based dynamic binary analysis and network-based informed probing. Compared with existing solutions, it has several unique advantages. First, it is fast and active compared to existing passive detection mechanisms. Instead of waiting for actual attacks/control to happen, we can proactively detect the existence of malware. Second, it is very reliable in detecting the malware. While attackers can generate very different binaries for samples in a malware family, the underlying MCBs are still the same and they are typically unique for different malware families. This is because the attackers still want to control all the malware (in the same family) in the same way to make them easily manageable. The accuracy and robustness of using MCB in detection are comparable to traditional host-based approaches (they both use fine-grained binary analysis techniques), and it avoids a lot of network evasions. Finally, our approach is scalable to large network deployments. Since we only need one scanner for the whole network instead of installing detectors on every machine, the deployment, management, and MCB updating are relatively easy. It even provides the possibility of Internet-scale scanning/detection when necessary.

Specifically, our paper makes the following contributions:

- We propose a new detection strategy combining host-level dynamic malware binary analysis and network-level informed probing techniques. To the best of our knowledge, it is the first work to discuss (P2P) malware detection based on automated MCB (Malware Control Birthmark) extraction and informed network probing. It shows a novel combination of strengths (fast, reliable and scalable) from host- and network-based detection approaches.

- We develop PeerPress, a prototype system that implements the proposed framework. We design new techniques to determine if given malware opens a specific port and automatically extract the port generation algorithm/logic. Furthermore, we design new techniques to craft a specific/special MCB probing packet that can let MCB-enabled malware expose itself (from network observation perspective), much more efficient and effective than existing network software fuzzing techniques [7]. In particular, we develop ICE (*Informed forCed Executing*), a new technique to quickly identify possible MCB execution paths that can be used later in stitched dynamic symbolic execution to derive satisfiable packet contents to trigger MCB logic. In evaluation, ICE can save up to 80% overhead compared to traditional multi-path exploration schemes.

- We evaluate PeerPress with multiple representative and complex real-world malware families (including Storm/Peacomm, Conficker and more recently Sality). PeerPress successfully extracts their MCBs and demonstrates that using MCB-informed active probing, we can detect those malware infected machines with 100% accuracy and 0 false positive (in three /24 networks). It is able to detect these malware in an early stage when other tools (e.g., BotHunter) can not.

- We extensively discuss the limitations and implications of our approach (Section 7). While not perfect, PeerPress works great when MCBs are successfully extracted. Furthermore, even in the worst case (not able to penetrate into some malware binary), PeerPress can still use several special types of MCBs (e.g., "no response") to help recognize suspicious malware infections. We consider PeerPress as an important further step toward *proactive* malware detection and a great complement to existing passive detection techniques.

2. APPROACH OVERVIEW

2.1 Problem Definition

Assumption. We assume that a captured malware binary P is available, and we analyze it in our host-based analysis phase without source code access. With the wide deployment of honeypots to collect malware samples, this is a very basic assumption for most malware analysis and defense research [16, 36, 37, 42, 54]. Furthermore, since most malware binaries are now protected against static analysis (e.g., using obfuscation/polymorphic techniques), we mainly employ dynamic analysis techniques in this work.[1]

Since we target P2P malware, without loss of generality, we assume the malware sample P contains two independent program logics:

- P_1, which opens a network service port ψ.

- P_2, which parses certain network request(s) ρ and generates response(s) η through the network port ψ.

We assume all binary samples within the same malware family/version share the same and unique P_1 and P_2. These two program logics provide a remotely accessible/controllable mechanism that we capture as the *birthmark* of the

[1]Note that combining static analysis will definitely improve our approach.

malware family, which we call *Malware Control Birthmark* (MCB) in this paper.

More formally, a MCB can be defined as a pair:

<Portprint$\{P_1, \psi\}$, MCB probing ρ and response η >

Here *Portprint* denotes the service port(s) ψ used by the malware and the corresponding algorithm/logic P_1 to generate such port number(s). *MCB probing* denotes some well-constructed probing packet(s) ρ that trigger(s) the execution of malware control logic P_2 to reply with some (network observable) unique response(s) η.

2.2 Approach Overview

We illustrate the overview of PeerPress in Figure 1.

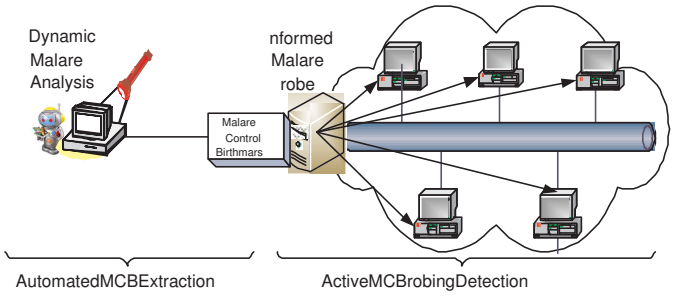


Figure 1: Our Two-phase Approach: PeerPress

The first phase is automated MCB extraction through dynamic malware analysis. In this phase, we analyze the malware sample and extract its MCBs (including both Portprint $\{P_1, \psi\}$ and MCB Probing ρ, η) if possible.

- Portprint extraction. To identify a portprint, we first run the malware P in a test environment and collect the trace from the malware starting up to opening a socket and binding this socket to a port. We capture the network service port ψ and further reason about the generation of such port. If the port number is environment dependent and/or algorithmically generated, we need to further extract its generation logic P_1.

- MCB probing extraction. Using the same analysis environment, we begin with sending random fuzzing packets to trigger the execution of logic P_2. Leveraging the basic execution trace, we perform *directed, informed* multi-path exploration to identify interesting MCB execution paths. We further employ concrete and symbolic execution techniques to derive MCB probing packets (input) ρ and the corresponding response η. To verify the uniqueness of MCBs, we examine ρ and η to ensure it is not the similar benign traffic targeting port ψ.

The second phase is MCB-assisted network probing. We will use our extracted MCBs to guide probing of networked computers to quickly and reliably identify malware infected victims. More specifically, targeting the P_1-generated port ψ, we employ a network scanner S to probe each host. If we observe the desired ρ and η pair from probing, we report the machine as compromised (by the specific P2P malware).

As we can see, the MCB-assisted network probing is relatively straightforward once MCBs are generated. In the rest of the paper, we will focus on the automated extraction of MCBs.

2.3 Key Challenges and Basic Ideas

Challenge 1: Extracting and reasoning about the dynamic portprint $\{P_1, \psi\}$. It is worth noting that the port number ψ that we might observe in the analysis environment may not represent the actual port number that will be opened on compromised machines. This is because the malware instance P interacts with different environments on different machines, which could influence the generation of ψ. One real-world example is the Conficker worm [45], which binds to different ports based on different IP addresses. Although we know that P generates ψ in the analysis environment E_t, we still need to derive the corresponding port ψ_i in the probing environment E_i when infected by the same malware. The dynamic attribution of the listening port on targeted machines represents a challenge.

We find that malware generates its listening port in three ways:

- Static. In this case, the malware always opens a fixed port number, which might be defined in a configuration file or is embedded in the binary. For example, Nugache [50] always listens on TCP port 8.

- Algorithmically deterministic. In this case, the malware uses some algorithm to generate a host-specific port number. This algorithm can take various parameters, e.g., IP address and time. Conficker.C belongs to this type [2]. We envision that more future malware might use this advanced feature because it removes the need of some central servers or super peers to collect port information and then coordinate/distribute among other nodes for bootstraping peer discovery in traditional P2P malware.

- Random. The malware listens on some randomly generated port. In this case, our probing scanner will have to utilize existing network traffic monitoring or port scanners to identify the opened ports on end hosts. With the widely deployed network monitoring and scanning tools already available to network administrators, this should not be a significant issue.

Thus, an effective solution should tell us the portprint type of a given malware program (static, random, or algorithmically deterministic). Furthermore, it should provide the port generation logic/algorithm P_1 (particularly when it is algorithmically deterministic) and the knowledge of the environment it depends on (e.g., IP/Mac address, machine name, or system time). In this case, given a new target machine i to scan, we can run the same portprint logic P_1, simulating environment e_i on machine i as the input parameters to generate the target port.

The problem of determining the type of portprints and the sources of portprints can be solved by using well-known taint analysis techniques [13,51,56]. However, different from most traditional *forward* taint analysis work [13,56] to solve known-sources-to-unknown-sinks problems, our problem is essentially *many-unknown-sources-to-one-known-sink*. Thus, we start from the port number and perform offline *backward taint analysis* to obtain the complete data dependence flow for the port generation. Based on the semantic meaning of the sources, we can determine the portprint type, and the necessary environment parameters that will contribute to the port generation. Furthermore, to extract the portprint

generation logic P_1 as an independent program, we apply classic backward program slicing techniques [40] in a similar way to related work [16, 36, 37]. In short, since our techniques on extracting portprint are mostly on top of existing work [13, 16, 36, 37, 51, 56], we put some detailed description in our extended technical report [55] and the rest of the paper will mainly focus on the second challenge.

Challenge 2: Efficiently Exploring and Extracting MCB Paths Inside P_2. Regarding the packet parsing logic P_2 inside P, we aim to find all possible execution paths that start from packet receiving routines (e.g., `recv()`) to packet transmitting routines (e.g, `send()`). This is a basic requirement for candidate MCB paths, because as we mentioned before, we assume a well-constructed MCB probing packet ρ can trigger a specific response η along a MCB path.

Thus, the problem becomes *how to efficiently find all possible MCB paths in P*? It seems that existing multipath exploration approaches [15,42] could be applied directly. However, these approaches typically follow a depth-first search scheme and randomly choose a path when reaching any branch point. As a result, if they are used in our application, they will blindly explore all possible (although mostly unnecessary) paths to find desired MCB paths. Compared with these traditional *trigger-to-unknown-behaviour* exploration model, our problem is better defined as *trigger-to-one-response* model. Essentially, the goal of traditional multipath exploration approaches is mainly to excavate dormant behaviour, which is quite different from our goal.

Our proposed solution, Informed enforCed Execution(ICE), combines both forced execution [54] and concrete/symbolic execution [15, 42, 51] techniques to improve the effectiveness and efficiency in finding MCB paths. During execution, ICE first takes a breadth-first search approach to quickly obtain an overview on the packet processing procedure before going into any depth (sub-functions). Furthermore, ICE employs *directed* search when exploring paths at branch points with the intuition that some paths containing certain functions/calls are more likely MCB paths. Examples of these functions include those that directly call `send()`, or indirectly call functions that wrap `send()` (several layers of wrapping is possible here). We specifically define *function containers (FC)* to refer to such functions that when called they will reach our desired network routines such as `send()`. Code blocks leading to those *FC*s that end with valid network transmission such as `send()` are preferred when exploring paths. Moreover, a special type of *FC*s will denote functions that lead to network/process termination such as `closesocket()` and `exitprocess()` without sending out network information. Code blocks leading to these *FC*s should be given lower priorities. Basically, ICE automatically creates and maintains the list of different *FC*s and uses them to make the best possible decision at any branch point. When exploring new paths at a branch point, ICE has a *Foreseeing* step to analyze the next k code blocks to decide the priority of branches to take. Generally speaking, ICE will prefer the branch containing high priority *FC*s and then force the execution towards that path. We discuss the detailed exploration algorithm in Section 3.

3. MCB PROBING EXTRACTION

3.1 ICE: Efficient Path Exploration

As stated earlier, the problem of finding candidate MCB paths is different from the problem of traditional multipath exploration. In particular, as discussed in previous work [21], path exploration without any high-level semantic guidance is inefficient. In our context, we introduce three novel exploration guidelines to efficiently identify MCB paths: (1) Enlarge the sinkhole hit range using *Function Containers*; (2) Make wise decision on branch points by *Foreseeing*; (3) Complete the MCB path through *Stitched Dynamic Symbolic Execution*.

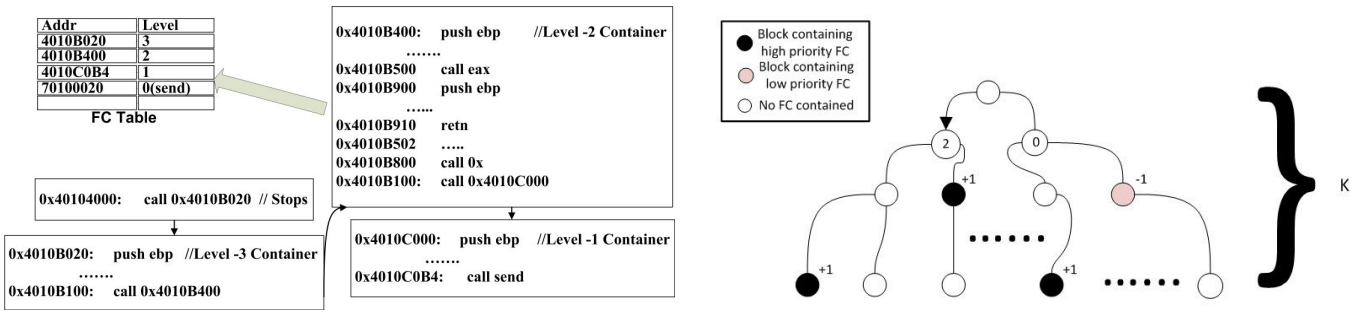
Collecting Function Containers Inside Malware. Blindly (randomly) exploring paths inside malware is not efficient in our context. Thus, we employ *directed* path exploration techniques for finding candidate MCB paths. Since a MCB path has some desired patterns, e.g., typically containing a sinkhole point of network transmission routines such as a *send* library call, it makes sense to choose paths that likely reach these sinkholing routines. In particular, to expand such limited small number of sinkholing routines to a larger hit surface, we introduce the concept of Function Containers to assist directed exploration.

Definition *A function container is a function satisfying any of the following conditions:*
(I) Any desired sinkholing system/library calls are automatically function containers, i.e., $SysCall_{desired} \in FC$*;*
(II) The function directly or indirectly contains/wraps an existing function container. Furthermore, the call of this FC will lead to the call of $SysCall_{desired}$*.*

In this definition, $SysCall_{desired}$ refers to interesting, critical system/library calls that will be typical sinkhole points, e.g., `send()` and `closesocket()`. Condition II implies that one FC can be wrapped by another FC, i.e., FCs can have multiple levels. One example of Conficker's send-out routine is shown in Figure 2(a), which illustrates four separate FCs (at different levels). Condition II also implies that the call of a FC will invoke the desired system call. Although this is very hard to verify without source code or full static analysis, in this paper, we approximate this condition if it holds in *all* our recorded dynamic traces.

Function containers can be analyzed with current static analysis techniques but are harder to completely construct based on dynamic analysis only. To initialize our FC table, we take advantage of the malware execution traces that our malware analysis environment generates. Often, we find that malware sends out packets to contact peers and initialize its membership in the P2P network. This initial activity is valuable, because the networking code used for these packets is typically the same that is used by the P2P service logic. Thus, these initial traces typically allow us to determine which functions are used for network traffic. Conficker is one typical example. For each trace, we collect both control and data flow information to conduct the automatic offline analysis. If we find our desired system/library calls, we trace back the call frames and extract a set of n level containers and record them into our initial FC hashtable.

During the online path exploration, we follow a breadth-first principle and enforce the execution towards code blocks containing high priority FCs (e.g., those will lead to network



(a) Different Levels of Functional Containers (b) Path Foreseeing

Figure 2: ICE Illustration

transmission routines). At the same time, we also update our FC hashtable if our initial FCs collection is not correct or not complete. We use two policies to update the FC hashtable: (1) If one trace shows that after entering a certain FC the trace does not lead to the desired system/library call, we delete it and its upper level FCs from the FC list (since it violates condition II);[2] (2) If we find one critical system call executed but not yet defined in the FC hashtable, we create a new set of level-n containers for this system call.

Foreseeing. As shown in Figure 2(b), our dynamic analysis needs to make decisions at each branch point to determine which path to take/prefer. We leverage *Foreseeing* for this purpose.

In detail, we foresee (statically look forward) k code blocks to search for the calls to any recorded function container. As seen in Figure 2(b), if a high priority FC is contained in a code block, ICE assigns a priority score of $+1$ for the block. Similarly, it assigns -1 in the case of encountering a low priority FC. Then, ICE simply sums up the total priority scores Λ among all code blocks in the $Left$ and $Right$ branches and gives preference to the branch with the overall higher priority score. We enforce the branch decision [54] at such branches and repeat the foreseeing till we hit a target FC. To prevent exploring the same path again, we set the code block that we have explored as low priority. For the case that priority score $\Lambda_r = \Lambda_l$, the exploration follows the natural execution choice. However, ICE will remember the decision point and go back to explore the other branch later.

When trying to find a new MCB path (a path from a *receive* to a *send* function), ICE starts from the snapshot at the *recv()* call and obtains the unexplored path information from a decision queue, which saves all the explored and unexplored branch information. ICE continues until (1) the queue is empty, (2) or a user-defined threshold θ of maximum MCB candidate paths is reached.

Loops and Indirect Jumps. One challenging issue of ICE is to handle control flow constructs such as loops and indirect jumps. For indirect jumps, our foreseeing operation may fail to predict the possible target. For loops, the priority score may be inappropriately set, which might lead to incorrect MCB paths. Thus, we need to detect such control flow logic and take special handling. Whenever our online execution module detects that one possible branch goes back to recorded addresses (loop) or jumps to certain variable addresses, e.g., `jmp eax`, we stop foreseeing and let the program execute naturally.

However, the basic execution may not work properly at all if the target address of an indirect jump or the number of loop iterations is incorrect (inconsistent) due to the previous forced execution. Thus, we have to perform special handling for loops and indirect jumps in ICE. For indirect jumps, we need to determine whether the probing packets contribute to the generation of the jump target. To do that, we perform concrete and symbolic execution. More specifically, we treat each byte of the probing packets as a symbol and track its propagation. If these bytes are used in indirect jumps, e.g., propagating to `eip`, we deduce symbolic equations at the point of indirect jumps. We will try to solve the symbolic equation and enumerate the possible target addresses. Then, we switch back to our online execution mode, analyze these possible target addresses, filter out impossible branches (if disassembled instructions at these addresses are invalid), and continue the execution to explore further paths. Similarly, for loops, we try to figure out whether symbolic input bytes are propagated to the loop counter. If so, we adopt a similar idea introduced in [47] to perform symbolic execution.

Stitched Dynamic Symbolic Execution. Because of our breadth-first exploration scheme, some MCB paths returned by ICE are probably not complete, with some functions not fully explored (in depth). Our next step is to complete the full MCB path by adding (or stitching) back these unexplored *sub-paths*. It turns out to be not a straightforward issue. This is because when entering these sub-paths, there might be again multiple different paths to explore (as illustrated in Fig. 3), and it is likely that only one (or some) execution path(s) can correctly (consistently) stitch back to the main upper layer MCB path (others will lead to other paths that deviate from the MCB one).

Another issue we need to handle is to filter invalid MCB paths returned in the previous step. These paths will never be actually executed by receiving probing packets but they are generated as artifacts due to the *enforced* execution. Thus, we need to filter them.

Both issues mentioned above are related to identifying correct MCB (sub-)paths and discarding irrelevant ones. We solve both issues using a combination of concrete and sym-

[2]Note that the program execution after entering the FC is nature (i.e., not enforced) because we need to update the FC tables based on whether the natural execution of FC directs to the target system call or not.

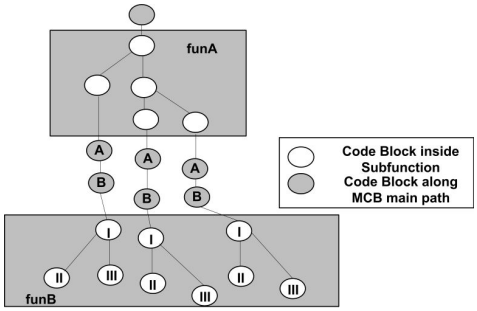


Figure 3: Stitch sub-paths along MCBs

bolic execution. One important assumption is that we consider the probing packet ρ as the only causal factor to drive P_2 to generate the response packet η along MCB paths. Thus, we consider a (sub-)path of MCB as valid when the relevant branches have control flow dependences on symbol variables in ρ. Specifically, for each MCB main path, we mark the size and each byte of the received buffer as symbol values. Then, we monitor the propagation of these symbols along the path. If we encounter an *enforced* path with no control dependence on the symbol variables, we discard this MCB candidate because its execution may not be dependent on probing packets.

When we encounter any function call in unexplored sub-paths, we provide the call with the current symbol context and create a snapshot at the entrance point. Thus, after we enter the function, we repeat the procedure mentioned before to keep track of the control flow with symbolic variables. When we encounter a branch possibly affected by some symbol variable, we record this address into a queue. It tells us that this branch has two possible choices and we need go back to the snapshot and explore another branch later. This step recursively occurs (in some cases, we need to go even deeper into sub-functions) until we find out all possible sub-paths along the main MCB path.

Handling Encoding Function. One special case needs to be further mentioned: the encoding function such as encryption, hash and checksum. Thanks to previous research [18,51,53], we can prevent infinite symbolic execution for such functions that possibly exist along the MCB path. We build an automatic tool to identify and bypass encoding functions using similar heuristics mentioned in [18,28,39,56]. To actually reverse such functions, we leverage partially automatic techniques similar to previous work [18]. We skip details here as they are not our major contributions, and we will discuss some limitations of current design in Section 7.

Solving the MCB path. When we obtain a complete MCB path (including sub-paths), the final step is to solve the constraints along this entire MCB path. The complete MCB path includes the information about symbol propagation. We construct the symbolic equations and input the resulting equation into a solver. If the equations are solvable, we can construct a MCB probing packet ρ that follows the MCB from the receive function to the send function and generates a response η.

Safe MCB Probing. Among all MCB probing packets, there is one type we refer to as *Safe MCB Probing*. Simple examples of this kind of packets include error messages that are mainly triggered when there are wrong commands/operations. A safe probing packet implies that the malware does not execute any problematic (dangerous) operation along this path but already exposes itself in a unique way. In our implementation, we simply identify the length of the corresponding trace and system calls recorded in the trace. If these traces contain no dangerous system calls and the trace length is shorter than others, we consider this corresponding packet as potential safe MCB probing.

Limitations. ICE suffers from some of the problems in existing multipath explorations approaches [15,42,44] due to the dynamic analysis nature. For example, one issue leading to some undiscovered control flow is shown in Fig. 4. If malware has different functions (A and B) to parse packets, based on different system configurations, e.g., `configure_A` and `configure_B`, ICE may only generate one possible MCB path that is observed during the execution and thus fail to probe machines with the other configuration. This problem could be solved by taking into account environment-sensitive/implicit control flows, which we leave for future work.

```
1   If configure_A==TRUE:
2       process_A(received_packet_buffer)
3   If configure_B==TRUE:
4       process_B(received_packet_buffer)
```

Figure 4: Example of Undiscovered Control Flow

3.2 Verifier: Filtering False Positive Cases

After the dynamic analysis of the malware sample, we need to verify whether our generated MCB can actually be used for probing. That is, we have to ensure that the probing packet can actually trigger the malware to reply uniquely. We run the malware sample again in another clean environment without any instrumentation and check whether it responds to the probing packet with the expected reply or not. If so, we go to the second round verification.

Second round verification is to verify whether the reply is *unique* or not. We need to ensure that the response is not the same as a response from some normal well-known (P2P) software. For that purpose, we need to manually build a whitelist database of multiple, well-known benign services/applications/protocols, including most well-known P2P software, for their normal request-response patterns. Such patterns include identifying the protocol specification and marking all the fixed/variable fields inside the response. In this way, we can search the whitelist and find whether our extracted MCB is a unique evidence or not.

Limitations. In reality, it is very challenging and almost impossible to construct a complete and precise database as whitelist. In our preliminary implementation, we simply collect around 50 benign P2P/FTP/HTTP software including Apache, FileZilla, eDonkey, eMule, Morpheus, Limewire, Kazaa, which we intend to expand over time. We test each MCB probing on these benign software and make sure they will not generate the same response as the malware does. For each benign software, we carefully read the related protocol documents (e.g., eDonkey P2P, FTP) and extract the

patterns mentioned before. Clearly, this manual work is tedious and may not work for benign software using unknown or undocumented protocol. In this case, existing automatic protocol reverse engineering techniques [17, 20, 23, 24] could help us to build models for legitimate protocols, and we can then use these models to improve our whitelist. This is especially important to find minor differences between MCBs and benign protocols. We also note that false positive cases are possible due to the incompleteness of the whitelist. However, the incompleteness will mainly cause possible false possibles but not false negatives. Nevertheless, we did not see false positives in our experiments. We discuss more implications in Section 7 and leave it as future work to construct a better whitelist database.

4. IMPLEMENTATION

Our PeerPress implementation combines both (online) dynamic analysis and offline processing. For online analysis and recording, we implement two versions based on toolset `DynamoRIO` [3] (which is lightweight but may fail on some malware) and `TEMU` [9] (which is more robust to run malware but is also more heavyweight). Online modules take charge of generating instruction traces and performing informed, forced execution. As an illustration, for our TEMU-based implementation, we have developed three new independent plug-ins. The first is to record fine-grained (for port generation logic and probing packet parsing analysis) and coarse-grained (for FC extraction/update) traces. The second is to enforce certain branch decisions based on the input of the offline analysis components. It executes the foreseeing operations by disassembling code blocks following undecided branches and searching for the calls of FCs. The last plug-in is for general execution control. It helps us to start execution at specific addresses, dump/modify memory/registry values, create execution snapshots and perform system/library hooking. Furthermore, we enhance TEMU's taint analysis to support tainting the input/output of specific calls.

Our main implementation effort concentrates on offline modules. Our offline modules are mainly built using `Python`, and they include program dataflow analysis, program slice generation, and ICE input generation. For example, in portprint extraction, we perform backward taint analysis and program slicing on fine-grained traces. At the same time, our offline module collects semantic information to derive the portprint type and to extract the program slice for online replaying. In ICE, to support multi-round exploration, our offline module first fully analyzes the trace we generated in the previous round. Then, it provides our online module with a concrete specification of the path exploration. It includes where to start execution, whether to create snapshot or not, what branch decision chain for our online enforced execution to follow first, and how to modify the memory/register value. Furthermore, for symbolic execution, we developed a module to translate the instruction traces into the `VEX` intermediate language using `libVex` [4]. We developed our own symbolic execution engine and an interface to `Z3` [11] constraint solver.

5. EVALUATION

In this section, we evaluate PeerPress on several real-world malware families, which are listed in Table 1. This includes representative and complex modern P2P bots such as the infamous Nugache malware [50], Phatbot, Storm/Peacomm [48], Conficker C [2], and more recent Sality [12](still active in the wild as the writing of this paper). We also include several Trojan horse/backoor malware, because they also contain MCBs (many of them could also be considered as bots). This is to further demonstrate that PeerPress can detect more than just P2P malware, as long as PeerPress can extract MCBs from the malware. These malware samples were collected from multiple online malware repositories such as [1, 5] and diverse security researchers. We verified the ground truth labels of these malware with multiple online malware analysis services such as [1, 10] and manual examination on binaries and network traffic.

5.1 Effectiveness of Portprint Extraction

We extracted portprints for each malware family and we summarize them in Table 2. To verify their correctness, we run these malware multiple times in a clean environment and each time compare our extracted portprint with the actual port the malware bound to. The detailed result is shown in Table 2.

Among all the malware we have examined, Conficker C has a complex and unique port generation logic, which was previously manually analyzed in [45]. Now with PeerPress, we can automatically extract this logic within a few minutes. Furthermore, PeerPress provides a clear function interface with parameters and their semantic meanings because it captures system calls such as `getpeername` that parse the buffer related to the slice arguments. It is worth noting that algorithmically deterministic portprints are a strong evidence of the malware existence. That is, with only portprints (even without further MCB probing packets/response), we can already detect this kind of malware with very high confidence.

We find that many portprints are static in our tested malware. Most of such malware embeds the port number in the binary, such as NuclearRAT and NuCrypt, or reads from some configuration file, such as the case of Peacomm/Storm. Only a few malware samples (Phabot and Nugache) listen on totally random ports. In our tests, the ports were used for FTP services in both cases (to provide egg downloading service for newly infected malware). This inspired us to probe suspicious random ports just using an FTP packet and monitor their reply. In Section 5.3, we further demonstrate even though the malware may use the standard FTP protocol, the slight implementation differences may still expose themselves. One very interesting case is the algorithmically deterministic portprint of Sality (UDP port), because previous reports have claimed that the port is selected pseudo-randomly [12]. We carefully examine our generated portprint and find that there are two source bytes that are the result of system call `GetComputerName()`. These two bytes are multiplied, and the result is added to a constant number `0x438`. Meanwhile, through tracking the control dependences, PeerPress also successfully extracts another path which forces the malware to bind to a static port, 9674. We deduce that the reason why security reports such as [25] claim the port is psedo-randomly generated may be because: (1) The computer name can be considered as a random value. (2) It is possible for malware authors to reconfigure the constant number `0x438` to other constant value. PeerPress declares that the portprint of Sality is algorithmically deterministic, and it extracts the

Name	*Type*	*Name*	*Type*
Conficker C [2]	P2P Bot	Nugache [50]	P2P bot & Trojan Horse
Phabot [6]	P2P Bot	Sality [12]	P2P Bot
Storm/Peacomm [48]	P2P bot	BackOrfice	Trojan horse/backdoor
NuclearRAT	Trojan horse/Spyware	WinEggDrop	Keylogger/Spyware
Penumbra	Backdoor	WinCrash	Backdoor
NuCrypt	Trojan horse/worm	Wopla	Trojan horse

Table 1: 12 malware families in our evaluation

Malware	*Type detmined by MProbe*	*Observed Port Number*	*Description*	*Correctness*
Conficker C	algorithmically determined	46523/TCP and 18849/UDP	Program Slice with IP and time	✓
Nugache	static/randomly generated	8/TCP, 3722/TCP	Open Multiple (fixed/random) Ports	✓
Sality	algorithmically determined	6162/UDP	Generated based on Computer Name	✓
Phabot	randomly generated	1999/TCP		✓
Peacomm	static	7871,11217/UDP	Read from spooldr.ini	✓
BackOrfice	static	31337/TCP	In binary	✓
NuclearRAT	static	190/TCP	In binary	✓
WinEggDrop	static	12345/TCP	In binary	✓
Penumbra	static	2046/TCP	In binary	✓
NuCrypt	static	3133/TCP	In binary	✓
Wopla	static	8080/TCP, 25099/TCP	In binary/file	✓
WinCrash	static	1596/TCP	In binary	✓

Table 2: Portprint details of different malware families

program slice with the target computer name as the parameter. Once provided with computer names (which should be available to most network administrators), PeerPress can probe target machines to detect Sality infected victims.

5.2 Effectiveness of ICE

In this section, we evaluate the effectiveness of ICE. First, we conduct an experiment to verify that there are multiple function containers in each malware binary, which supports our assumption that function-level abstraction is feasible in dynamic analysis. Second, we verify that it can significantly reduce the overhead of path exploration compared to existing exploration scheme.

Function Containers in Malware Binary. In our evaluation, we set the maximum call depth level as 4, and locate on average 28 function containers per malware sample using this level. In our tests, all containers eventually lead to desired system calls. More interestingly, throughout all our test cases, malware calls these containers if they want to execute specific tasks.

Overhead Comparison. To evaluate whether our informed execution can efficiently locate desired MCB logic, we compare the performance of ICE with the traditional approach that randomly chooses a path to explore next [54]. Here, the performance is measured using the *number of rounds* to find all MCB paths (that the system succeeds in finding using a brute-force approach), and each round is defined as one path exploration attempt from the sink (receiving the probing packet) to the end of the execution run for this path. Note that we do not claim to be able to explore *all* execution paths in the program. Instead, our baseline of all MCB paths is determined by brute-force exploration of all possible paths that can be directed/triggered by one single probing packet (i.e., we may miss MCB paths that can only be triggered by multiple probing packets). All these paths start from packet receiving till (i) the malware sends out some response, or (ii) the communication/process terminates. In this test, it is not very important whether we obtain accurately all MCB paths or not. Instead, more importantly we want to see which technique is quicker to locate these MCB paths given as the baseline. The result is shown in Figure 5. We can clearly see that our ICE significantly outperforms the traditional forced executions [54]. Our method requires much fewer exploration rounds to find MCB paths. In many cases, our system reduces the overhead up to 80%.

5.3 Overall MCB Extraction

PeerPress successfully extracts on average about 6 MCB probing/response pairs per sample from all the tested malware, as shown in Table 4. In terms of running time, we select three most complex, representative malware samples and report the performance for different components of our system in Table 3 (performance of other samples are similar or better, omitted here due to space limitation). We acknowledge that some steps, such as semantic derivation and symbolic execution are relatively slow, which is not surprising considering that we are analyzing very complex real-world P2P malware in a fine-grained way with some known-expensive operations. Compared with existing state-of-the-art work (e.g., [17, 18]) that also uses expensive dynamic analysis and symbolic execution techniques, our performance is on par with those studies, and we believe it is reasonable and tolerable for offline analysis of malware families (recall that the analysis does not need to be repeated for each individual sample). It can certainly be improved by optimizing our code, parallelizing some operations, and using more powerful hardware.

Among all MCBs that PeerPress extracted, the simple case is represented by certain Trojan horses/backdoors that provide some unique "Welcome" information in their response. It is actually a very effective and safe MCB without much effort to generate. We can initiate connections to the suspected host and verify whether it welcomes us in the specific way or not. This welcome message is most common in old fashion Trojan horses, because an adversary may use any re-

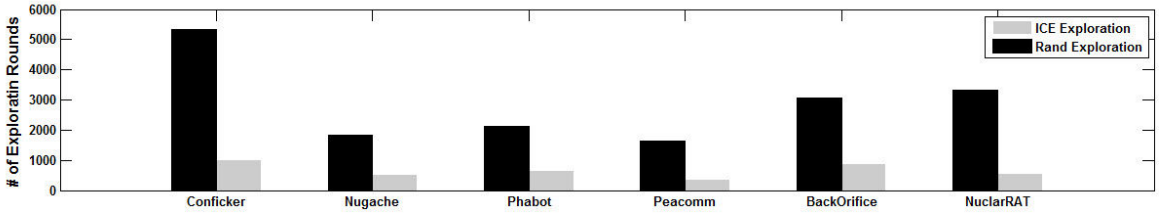


Figure 5: Performance comparison of ICE and Random Exploration

	Conficker C	Nugache	Peacomm
Fine-grained Recording (min)	38	21	37
Backward Taint (sec)	243	549	780
Program Slicing (sec)	180	363	173
Semantic Derivation (sec)	2813	489	541
ICE engine (sec/trace)	54.4	38.9	40.3
Symbolic execution (sec/-trace)	6863	1602	2711

Table 3: Running time of MCB extraction

Malware	*# MCB*	*Malware*	*# MCB*
Conficker C	3/3	Peacom	6/3
Sality	1/1	BackOrifice	16/14
Phabot	13/9	NuclearRAT	17/12
WinEggDrop	11/8	Penumbra	16/13
Nugache	21/7	WinCrash	1/1
NuCrypt	2/2	Wopla	2/2

Table 4: Statistics on extracted malware MCBs. (Here X/Y in Column *# MCB* means there are X candidate MCBs and Y final MCBs after verification.)

mote client to control the bots. We find this in Nugache FTP logic and some other malware, e.g., *WinCrash* and Wopla. For example, Nugache uses the following welcome message: `220-220-Welcome 220`.

We find that many MCB probing packets are easy to craft because there are no (or not many) encoding routines. More precisely, we found cleartext FTP logic inside Nugache, Wopla and Phatbot, peer synchronization logic inside Peacomm, and command and control logic inside traditional Trojan horses. Even though there are only a limited number of samples, our system is robust and fast to obtain MCB probing in a fully automatic way. In detail, we find one simple FTP service logic hosted by Nugache on a high-order port. After traversing the MCB paths, we extract 21 command and response pairs. After further verification, 14 are filtered (e.g., command `ls` and `pwd`) because they are not be considered as unique evidence. As an interesting MCB example among the rest, we find that the Nugache FTP service needs users to provide `Username` and `Port` for validation, which are quite different from normal FTP services we see.

For Peacomm/Storm case, PeerPress extracted six MCB probing candidates. We test these MCBs on the benign `eDonkey` clients and filter out three. One filtered example is a 509-byte probing packets (with the first two bytes as `0xe3 0x13`) that will receive a 18-byte response packet beginning with `0xe3 0x14`. This is actually used for regular peer recognition in the `eDonkey` protocol. The remaining three are interesting MCBs, include a probing packet beginning with the first two bytes `0xe3 0x0d` and the corresponding response packet beginning with `0xe3 0x0a`.

For the most sophisticated cases, we have to bypass the encoding function before the symbolic execution. As described before, we apply a semi-automatic approach to extract the encoding function inside of the traces. We automatically locate the `RC4` encryption and checksum routine inside Conficker and Sality, using several heuristics including highly-mixed receiving buffer [18]. We also successfully identify two double-word decryption keys inside the Conficker and Sality packet (with payload offset 2 and 0). Thus, we can recover the encrypted probing packet after the symbolic execution. Examining the cleartext payload, we find one key data field containing the payload version inside both Conficker and Sality. Both malware programs generate replies if the received payload version is lower or equal to its own binary version. It seems that the P2P logic implements a self-updating procedure, and the only way to trigger its reply is to provide a payload with a suitable version number. Another interesting finding about the Sality botnet is the *double replies*. When we feed our probing packet, Sality sequentially replies with two packets. One packet attempts to start a new UDP session while the other one is a reply to our MCB probing.

Although PeerPress extracted MCBs from all tested malware, we note that it does *not* mean PeerPress can extract *all* MCBs inside malware. We actually encounter some issues due to some complex control logic inside some malware programs. For example, PeerPress failed to extract MCBs from Nugache's port 8. We find multiple $WaitForSingleObject$ calls in the traces, waiting for some (asynchronous) event from other threads/process. ICE failed to correctly explore the paths in that situation. In the case of Conficker, PeerPress is not able to automatically crack the multi-round advanced encoding routines, thus failed to extract MCBs on some ports. The fact that PeerPress failed in several cases is not surprising, as we are dealing with real-world complex malware. However, our results are still encouraging because PeerPress could extract at least one meaningful MCB for all families that we examined.

5.4 Detection Results through Active Probing

In this section, we conduct the experiments to verify that our MCB-informed active probing can detect our targeted malware in a *reliable*, *robust*, *fast* and *scalable* way.

Test in Virtual Networks. We built one virtual environment with six virtual machines. All virtual machines installed Windows XP SP1 without new patches. We ran-

domly selected two different malware samples (from Table 1) to install on each machine (and eventually cover all twelve malware families in six VMs). Meanwhile, we installed some well-known benign services, such as Apache web server, P2P clients (e.g., edonkey), and FTP servers (e.g., Filezilla). Our probing engine uses extracted MCBs to actively probe the entire virtual network. PeerPress correctly detected all the existing malware in the virtual network without false positives. In terms of detection speed, it only took on average 1.103 seconds to detect each malware. This demonstrates that the informed active probing is an effective approach to detect malware in the network.

To further verify the robustness of PeerPress to detect different variants in the same malware family, we further collected three additional (but different) binaries of the same malware for Conficker, Storm/Peacomm, and NuclearRAT, and Nugache, respectively[3]. Our test environment is the same as mentioned before. PeerPress can not only detect all the variants but also correctly classified all variants into its original families. This again verifies that MCBs are unique for the same malware family and PeerPress is robust in detecting different malware variants in the same family.

False Positive Test in Real Networks. Next, we scanned our campus network (we randomly choose three /24 networks with no firewall to filter our scans) to test the real-world performance of PeerPress using the above extracted MCBs. We did not find any false positive during the scan, because most hosts do not have the corresponding (malware portprint specific) ports open. This is not surprising because our campus networks/computers are well managed/secured. We then intentionally scanned other open ports on these machines in order to further test the false positive of using MCB probing/response. We chose to scan port 80 (web) and all ports above 1025 in these three networks in hope to find some P2P applications. We found 58 hosts opened port 80 and 110 hosts opened higher ports, varying from several well-known P2P ports such as 6881 (BitTorrent) and 49153-49156 (uTorrent/Azureus) to some unknown ports. Our MCB-informed probing again did not yield any false positive. The probing speed for each host is about 1.128 seconds on average per MCB (including the first TCP port scanning interaction and the following MCB probing packet/response). Considering that it is easy to perform parallel scanning using multiple threads, PeerPress demonstrates good detection speed/scalability.

Comparisons with State-of-the-Art Detection Systems. In terms of an efficiency comparison with some state-of-the-art malware detection systems, we can mainly do a paper-and-pencil case study here because we could not obtain most of these tools. AccessMiner [38] is one relevant host-based detection system. It has a high accuracy and covers a lot of malware families. However, it may not be good enough at the stage where a P2P bot is waiting to receive commands from the botmaster, because it has not triggered its malicious logic yet. Meanwhile, it may also consume considerable resources on each end-host, so it is less scalable for deployment on large networks.

We further deploy another state-of-the-art network-based detection system, BotHunter [31], in our test (virtual) network and no malware (on six machines) is detected. This is reasonable because BotHunter needs to accumulate actual evidence related to multiple phases in the malware infection life cycle. In our cases, most of malware does not exhibit malicious network activity because the samples did not receive any commands. This also exposes one common limitation of many existing detection systems: they are passive and could be slow in terms of detection speed. On the contrary, PeerPress can actively detect those malware, even *before* those infected machine are accessed/controlled by remote peers/botmasters.

Note that compared with existing systems, PeerPress does have a limitation regarding to its detection scope. As clearly mentioned, PeerPress only targets malware that has MCBs, instead of all malware. However, we still consider it a valuable addition to our arsenal, because P2P malware and Trojan/backdoors are serious and emerging threats that we need to address. PeerPress greatly complements existing passive malware detection approaches.

[3]For these four malware we could find different binaries/-variants.

6. RELATED WORK

We now review additional related work previously not mentioned.

Multiple-path Exploration. One related research is the exploration of dormant functionalities [15, 42, 44, 54] in malware binary. In [42], the authors take snapshots at each branch point and reset when an additional branch needs to be explored. Wilhelm et al. [54] present a forced sampled execution approach to explore multiple rootkit execution paths. However, both exploration schemes still depend on random choice because they cannot correctly define what is the target function they want to explore. Our goal is to explore the MCB paths, so the exploration can be effectively accelerated and the overhead is significantly reduced. Meanwhile, ICE solves the problem of exploring the sub-paths along one explored MCB main path, which is different from the problem solved by previous work.

Protocol Reverse Engineering. Automatic protocol reverse engineering (PRE) research [17,20,23,24] discovers the semantic meanings of network protocols. However, these studies were mostly focused on analyzing legitimate network protocols. In such cases, it is easy to elicit a response from the application, simply by using a legitimate client that sends a valid request. We do not know how a valid request looks like; in fact, one key aspect of our work is to efficiently locate MCB execution paths, which determine the format of probe packets that can be used to obtain responses. Moreover, PRE systems are broader in the sense that they attempt to reverse engineer entire packet formats and state machines. This is fine for legitimate applications, but might be too brittle when applied to malicious binary code. Our technique, on the other hand, focuses on a specific problem (the extraction of inputs that trigger responses), and hence, can be more robust. In addition, we introduce the idea of dynamic portprints, a concept that is not considered by PRE systems. Finally, we note that better protocol knowledge is certainly helpful in both crafting better MCB packets and verifying/filtering false positive cases, as mentioned before. Thus, we consider these PRE techniques to be complementary to our work.

Network-based (P2P) malware detection. Network-

based detection approaches [22, 29, 31, 33, 34, 41, 43] attempt to inspect network traffic to detect some anomalous activities, patterns, or structures. Their weaknesses were already discussed earlier. In [30], Gu et al. present BotProbe that actively sends probing packs through IRC channels to separate botnet C&C dialogs from human-human conversations to detect IRC bots. PeerPress differs in that we accurately extract MCBs from malware binaries to probe them.

Code Reuse. Previous work [16, 36, 37] applies forward taint analysis and backward program slicing to extract interesting, relevant instructions as a stand-alone program. Our portprint extraction uses similar techniques to solve a specific problem. Different from [16], PeerPress extracts a virtual function from the whole program level instead of single function level, and the way of reusing/replaying of the code is different. Different from [36, 37], PeerPress uses backward taint analysis instead of forward analysis because in our context, we have a clear sinkholing point (the port binding event) but many unknown source points.

7. LIMITATIONS AND DISCUSSION

In this section, we discuss the limitations and implications of our solution.

A notable limitation of PeerPress is that it cannot craft correct MCB probing packets in the case of advanced encryption or certificate-based authentication, even though it could identify/bypass these routines. However, this is a common problem for *all* malware analysis tools that aim to provide meaningful (network) input to malware samples [18]. Malware could use this to verify/authenticate our incorrect probing packets and refuse providing any future response. However, even in this worst case, we argue that this kind of "no response" is indeed a special, suspicious, recognizable response that could be used in MCB probing. Furthermore, we note that our technique can still successfully extract portprints, and in many cases, the portprint itself is enough to detect/confirm the malware (without actually sending MCB probing content).

To evade portprint extraction, malware authors may intentionally delay the port binding until some conditions are satisfied, e.g., the time reaches some specific date. Indeed, it prevents PeerPress from discovering the port binding at first sight with the cost of decreasing the utility (in terms of accessibility) of the malware. This issue could be solved if we skip all the `sleep()` related functions in the monitoring and analysis.

To slow down the analysis of ICE, malware authors may intentionally include many (bogus) branches directly after the packet receiving. Even in such case, ICE is still faster than random path explorations.

Another possible evasion is to faithfully mimic a benign normal protocol behavior. First, this will increase the workload of malware authors. Second, if not implemented faithfully, the malware still could be fingerprinted due to the subtle differences from normal protocols, as studies in this domain have shown [14, 19]. If the malware authors choose to copy code from existing open source software in order to avoid differences in implementation, the code replication/-copy [52] could become another possible point of detection.

Finally, we note that *within its detection scope* (when MCBs can be successfully extracted), PeerPress is fast, reliable, robust, and scalable. We believe it is a great complement to existing passive detection techniques even though it is not perfect (just as any intrusion/malware detection technique).

8. CONCLUSION

P2P malware is an important direction for future malware. Current P2P malware detection remains insufficient. In this paper, we propose a novel, two-phase detection framework that seamlessly bridges host-level dynamic binary analysis and network-level informed active probing techniques. It can detect P2P malware and beyond, as long as the malware has MCBs. We developed new techniques such as ICE to tackle our research challenges, and we implemented a prototype system, PeerPress, to demonstrate the real-world utility. Our initial results are very encouraging. Although not perfect, PeerPress demonstrates an important step toward *proactive* malware detection and defense (instead of passive monitoring), a direction worth more attention from the security research community.

9. ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant CNS-0954096 and the Texas Higher Education Coordinating Board under NHARP Grant no. 01909. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation and the Texas Higher Education Coordinating Board.

10. REFERENCES

[1] Anubis: Analyzing Unknown Binaries. `https://anubis.iseclab.org`.
[2] Conficker C Analysis Report . `http://mtc.sri.com/Conficker/`.
[3] DynamoRIO . `http://dynamorio.org/`.
[4] LibVex . `http://http://valgrind.org/`.
[5] OffensiveComputing. `http://www.offensivecomputing.net/`.
[6] Phabot. `http://www.secureworks.com/research/threats/phatbot/?threat=phatbot`.
[7] Sulley. `http://code.google.com/p/sulley/`.
[8] Symantec Internet Security Threat Report. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.
[9] Temu . `http://bitblaze.cs.berkeley.edu/temu.html`.
[10] Virustotal. `https://www.virustotal.com/`.
[11] Z3 EMT Solver . `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.
[12] Cybercriminals Making Sality Virus More Complex. `http://www.spamfighter.com/Cybercriminals-Making\\-Sality-Virus-More-Complex-16068-News.htm`, 2011.
[13] Thanassis Avgerinos, Edward Schwartz, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE S&P'10*, 2010.
[14] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proc. of USENIX Security'07*, 2007.
[15] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenke Lee, Cliff Wang, and David Dagon,

editors, *Botnet Analysis and Defense*, volume 36, pages 65–88. Springer, 2008.

[16] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Proc. of NDSS'10*, 2010.

[17] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. of ACM CCS'09*, 2009.

[18] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proc. of ACM CCS'10*, September 2010.

[19] Juan Caballero, Shobha Venkataraman, Pongsin Poosankam, Min Gyung Kang, Dawn Song, and Avrim Blum. FiG: Automatic fingerprint generation. In *Proc. of NDSS'07*, 2007.

[20] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of ACM CCS'07*, 2007.

[21] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proc. of USENIX Security'11*, 2011.

[22] Baris Coskun, Sven Dietrich, and Nasir Memon. Friends of an enemy: Identifying local members of peer-to-peer botnets using mutual contacts. In *Proc. of ACSAC'10*, 2010.

[23] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol description generation from network traces. In *Proceedings of USENIX Security Symposium, Boston, MA*, August 2007.

[24] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. of ACM CCS'08*, 2008.

[25] Nicolas Falliere. Sality: Story of a peer-to-peer viral network. Technical report, 2011.

[26] Alexander Gostev. 2010: The year of the vulnerability . `http://www.net-security.org/article.php?id=1543`, 2010.

[27] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent Kang, and David Dagon. Peer-to-peer botnets: Overview and case study. In *Proc. of USENIX HotBots'07*, 2007.

[28] Flix Grobert. Automatic identification of cryptographic primitives in software. Master's thesis, Ruhr-University Bochum,Germany, 2010.

[29] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Proc. of USENIX Security'08*, 2008.

[30] Guofei Gu, Vinod Yegneswaran, Phillip Porras, Jennifer Stoll, and Wenke Lee. Active botnet probing to identify obscure command and control channels. In *Proc. of ACSAC'09*, 2009.

[31] Guofei Gu, Junjie Zhang, and Wenke Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *Proceedings of USENIX Security'07*, 2007.

[32] Guofei Gu, Junjie Zhang, and Wenke Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Proc. of NDSS'08*, 2008.

[33] Duc T. Ha, Guanhua Yan, Stephan Eidenbenz, and Hung Q. Ngo. On the effectiveness of structural detection and defense against p2p-based botnets. In *Proc. of DSN'09*, 2009.

[34] Márk Jelasity and Vilmos Bilicki. Towards automated detection of peer-to-peer botnets: on the limits of local approaches. In *Proc. of LEET'09*, 2009.

[35] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proc. of USENIX Security'06*, 2006.

[36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security'09*, 2009.

[37] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *31st IEEE Symposium on Security and Privacy*, May 2010.

[38] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: using system-centric models for malware protection. In *Proc. of ACM CCS'10*, 2010.

[39] Felix Leder and Peter Martini. Ngbpa: Next generation botnet protocol analysis. In *SEC*, pages 307–317, 2009.

[40] Andrea De Lucia. Program slicing: Methods and applications. In *1st IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[41] Reiter M. and Yen T. Traffic aggregation for malware detection. In *Proc. of DIMVA'08*, 2008.

[42] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[43] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: finding p2p bots with structured graph analysis. In *Proc. of USENIX Security'10*, 2010.

[44] P.M.Comparetti, G.Salvaneschi, E.Kirda, C. Kolbitsch, C.Krugel, and S.Zanero. Identifying dormant functionality in malware programs. In *31st IEEE Symposium on Security and Privacy*, May 2010.

[45] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. `http://mtc.sri.com/Conficker/`, 2009.

[46] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proc. of ACSAC'06*, 2006.

[47] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proc. of ISSTA'08*, 2008.

[48] Joe Stewart. Inside the Storm. `http://www.blackhat.com/presentations/bh-usa-08/Stewart/BH_US_08_Stewart_Protocols_of_the_Storm.pdf`.

[49] Elizabeth Stinson and John C. Mitchell. Towards systematic evaluation of the evadability of bot/botnet detection methods. In *WOOT'08*, 2008.

[50] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the storm and nugache trojans: P2P is here. In *;login*, 2007.

[51] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proc. of IEEE S&P'10*, 2010.

[52] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proc. ACM CCS'09*, 2009.

[53] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. of ESORICS'09*, 2009.

[54] J. Wilhelm and Tcker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.

[55] Zhaoyan Xu, Lingfeng Chen, and Guofei Gu. PeerPress: Fast and reliable detection of p2p malware (and beyond). Technical report, Texas A&M University, 2012.

[56] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow formalware detection and analysis. In *In ACM Conference on Computer and Communication Security (CCS)*, 2007.