Towards Next Generation Malware Collection and Analysis

Christian Martin Fuchs
Department of Computer Science
Technische Universität München (TUM)
Munich/Garching, Germany
christian.fuchs@tum.de

Martin Brunner
Martin Brunner Security
Munich, Germany
martin.brunner@x90.eu

***Abstract*—The fast paced evolution of malware has demonstrated severe limitations of traditional collection and analysis concepts. However, a majority of the anti-malware industry still relies on such ineffective concepts and invests much effort into temporarily fixing most obvious shortcomings. Ultimately fixing outdated concepts is insufficient for combating highly sophisticated future malicious software, thus new approaches are required. One such approach is AWESOME, a novel integrated honeypot-based malware collection and analysis framework. The goal of our collection and analysis system is retrieval of internal malware logic information for providing sufficient emulation of protocols and subsequently network resources in real time. If protocol emulation components are trained sufficiently, a larger setup could even allow for malware analysis in an isolated environment, thus offering side-effect free analysis and a better understanding of current and emerging malware. In this paper, we present in-depth information on this concept as well as first practical results and a proof of concept, indicating the feasibility of our approach. We describe in detail many of the components of AWESOME and also depict how protocol detection and emulation is conducted.**

Keywords-malware collection; malware analysis and defense

I. Introduction

Cyber crime has become one of the most disruptive threats today's Internet community is facing. The major amount of these contemporary Internet-based attacks is thereby attributed to malware, which is usually organized within a botnet in large-scale scenarios. Such botnet-connected malware is on their part utilized for infecting hosts and instrumenting them for various malicious activities: most prominent examples are Distributed Denial of Service (DDoS) attacks, identity theft, espionage and Spam delivery [6][24][39]. Botnet-connected malware can therefore still be considered a major threat on today's Internet.

Due to the ongoing spread of IP-enabled networks to other areas we expect the threat posed by botnet-connected malware to increase and moreover reach further domains in public and private life. Thus, there is a fundamental need to track the rapid evolution of these pervasive malware based threats. Especially timely intelligence on emerging, novel threats is essential for successful malware defense and IT early warning. This requires both, acquisition and examination, of current real-world malware samples in sufficient quantity and variety, commonly acquired through meticulous analysis of the most recent samples. The evolution of malware over time has led to the development of intensive obfuscation and anti-debugging mechanisms, as well as a complex and multi-staged malware execution life cycle [37]. Each phase may include numerous measures aimed at maximizing installation success and reliability. In order to comprehensively analyze the full life cycle, the malware under analysis must have unhindered access to all requested resources during runtime. While this could easily be achieved by allowing full interaction with the Internet, this is not a viable approach in setups, which are forced to consider liability issues.

In this paper, we introduce AWESOME [1], which is short for: Automated Web Emulation for Secure Operation of a Malware-Analysis Environment. It is a novel approach for integrated honeypot-based malware collection and analysis. The overall goal of AWESOME is to capture and dynamically analyze novel malware on a large scale. To identify trends of current and emerging malware, we aim to cover the entire execution life cycle. That is, we want to track malware communicating via both known and unknown (C&C) protocols in an automated way within a controlled environment. In order to minimize harm to third parties, malware should by default have no Internet access during analysis. The whole procedure intends to trick a sample into believing it is running on a real victim host with full Internet access.

II. Background and Related Work

A. Malware Life-Cycle

Due to the predominant economic motivation for malicious activities backed by organized cyber crime also the sophistication of malware and the respective propagation methods continuously evolved, hence increasingly impeding malware defense. Thereby, the invested effort and the achieved result must be in a reasonable relation for a professional attacker. Thus, we experience the phenomena of a moving target. That is, cyber criminals chose their targets and attack vectors according to the best economic relation and an ongoing paradigm shift towards client-side and targeted attacks has been witnessed in recent years [6].

Widely spread malware is most effectively managed within a botnet, therefore, a newly compromised host is still to become a botnet-member. Many findings from recent research indicate an ongoing specialization of the various groups in the underground economy offering ”Malware as a Service” and ”pay per install” schemes including elaborated models for pricing, licensing, hosting and rental [9][18][24][26]. This often includes professional maintenance, support and service level agreements for the purchasable malware itself as well as innovations in the maintenance of infected victim hosts.

Therefore, there is

1) one group specializing on the development of the actual malware,
2) a second group deals with the operation platform and the distribution of malware (i.e., to establish botnets) and
3) a third group focuses on suitable business models.

As a result, also the actual malware itself evolved with respect to obfuscation techniques and anti-debugging measures. Thus, current malware checks for several conditions before executing its malicious tasks, such as hardware resources of the victim host, Internet connectivity or whether it is executed within a virtualized environment [16][34][46].

With respect to the facts outlined previously and findings of related work [9][37] we model the execution life cycle of today's (autonomous spreading) malware as depicted in Figure 1 [5]. Particular stages within this life cycle may differ depending on the malware type and are accordingly addressed in separate work (e.g., the life cycle of Web-based malware has been analyzed in [40]). A common setting consists of three phases:

1) **Propagation and exploitation**
   Within this initial phase a worm spreads carrying a malicious payload that exploits one or multiple vulnerabilities. In this context, a vulnerability encompasses also the human using social engineering techniques thus possibly requiring user interaction. Furthermore, a vulnerability may be OS based (e.g., a flaw in a network service) or - more commonly - application based. The latter includes specifically vulnerabilities in browsers, their extensions (such as Adobe's flash), Email clients (e.g., attachment, malicious links, Emails containing malicious script code, etc.) and other online applications such as instant messaging clients.
   The malicious payload of the worm may instrument a variety of attack vectors reaching from (classical) buffer and heap overflows to recent return oriented programming techniques [47], while evading appropriate countermeasures such as address space layout randomization (ASLR), data execution prevention (DEP) and sandboxing [30]. After successfully exploiting a vulnerability, a shellcode is placed on the victim host, which gets then extracted and executed, including decryption and deobfuscation routines when necessary.
2) **Infection and installation**
   As a result of executing the injected shellcode, a binary is downloaded and placed on the victim host. This binary typically is a so-called *dropper*, which contains multiple malware components. It is supposed to disable the security measures on the victim host, to hide the malware components and to obfuscate its activities before launching the actual malware. It is synonymously referred to as *downloader*, which has the same features except that it does not contain the actual malware but downloads it from a remote repository resulting in a smaller size [37]. As there is an emerging trend that multiple cyber criminals instrument a single victim host for their malicious purposes also several droppers may be installed (in parallel) within this step.
   Once the dropper is executed it extracts and installs further components responsible for hardening and updating tasks. That is, they prepare the system for the actual malware using embedded instructions. These tasks include, e.g., disabling security measures, modifying configurations and contacting a remote site for updates ensuring that the actual malware is executed after every reboot and impeding its detection and removal. After the update site has verified the victim host as ”real“ and probably worth getting compromised, it provides the dropper components with information on how to retrieve the actual malware (e.g., via an URL) and updated configurations, when necessary. Again, this may include multiple binaries each representing a different botnet.
   Once downloaded, the malware is executed by the dropper component installing its core components. Finally, these core components remove all other (non-vital) components resulting from previous stages and the malware is operational.
3) **Operation and maintenance**
   Initially, the malware launches several actions, which are intended to directly gain profit from the victim in case the attacker loses control over the compromised host later on. Therefore, the malware harvests valuable information such as credit card numbers and all kinds of authentication credentials and sends it as an encrypted file to a remote server under the control of the attacker. Next, the malware attempts to establish a communication channel to the attackers command and control (C&C) infrastructure awaiting further instructions. These may include commands to launch different malicious actions but also maintenance operations such as retrieving updates, further propagation or even to terminate and remove the malware.

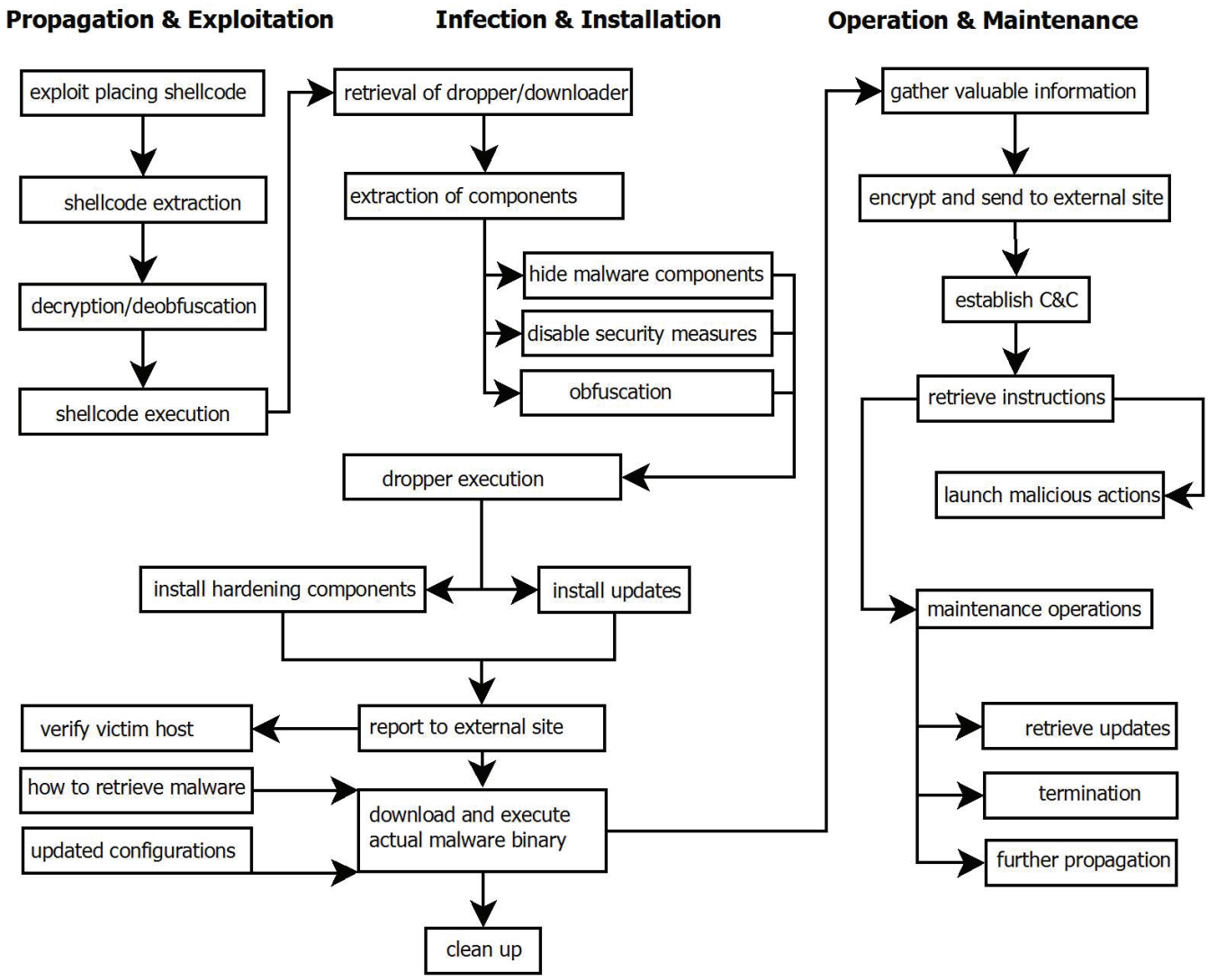

Figure 1. Today's multi-staged, complex malware life-cycle.

The various steps of the outlined, complex malware life cycle include many checks and measures to increase the resilience of the various features each intended to maximize the success of the malware installation, ensuring a reliable operation and to protect the cyber criminals from being tracked down. Thus, the reasons for this complex life cycle, especially the use of droppers in an intermediate step, are apparent:

- The dropper components evade discovery of system compromise. Also, the adversary has no need to distribute the core malware components in the first phase thereby impeding the successful collection and thus detection and mitigation of the malware.
- In addition, he can distribute the malware more selective and targeted.
- Finally, the attacker can ensure, that the victim host is real (i.e., not a honeypot or virtualized analysis system) and that it is worth being compromised (e.g., by checking available resources).

Among others, this implicates, that for a comprehensive analysis of a given malware it must receive all resources that it requests during its life cycle, since it may behave different or refuse to execute at all otherwise.

B. Combating Malware

One major issue that makes malware analysis a very challenging task, is the ongoing arms race between malware authors on the one hand and malware analysts on the other hand. That is, while analysts use various techniques to quickly understand the threat and intention of malware, malware authors invest considerable effort to camouflage their malicious activity and impede a successful analysis.

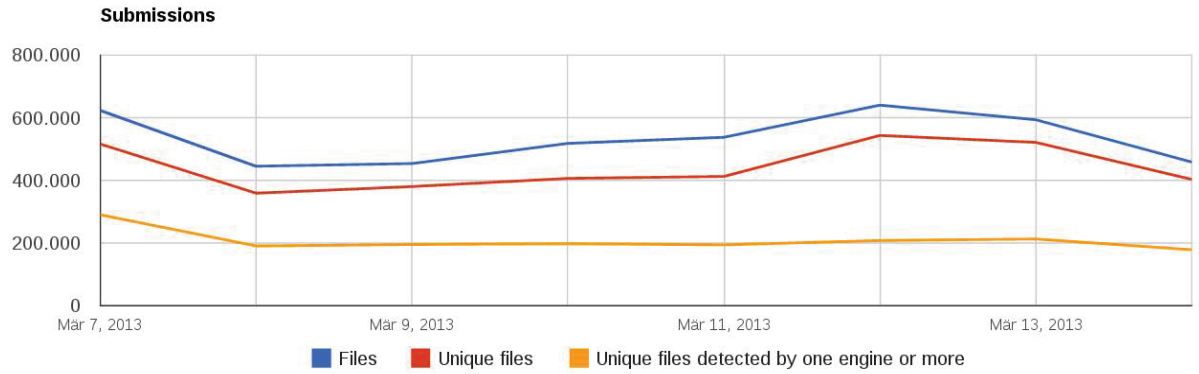

Figure 2. Amount of malware sample submissions to virustotal.com over a recent randomly chosen period.

While in past decades an explosive spread of identical malware has been observed, nowadays malware is mostly created utilizing creation kits. As a result, we experience a diffusion of malware variants, which are designed to be difficult to identify and analyze. Accordingly, a vast amount (i.e., hundreds of thousands) of new malware shows up *per day*. This is illustrated in Figure 2, which outlines the number of malware samples submitted to Virustotal based on a randomly chosen recent period.

Thereby the correlation between the total number of samples and the unique ones is notable, since they correspond to each other. This suggests that the majority of new malware can be attributed to be just new variants of already existing malware. That is, while most of the malware samples are unique (i.e., have different file hashes) they are in fact not. But since they are considered to be unique all of them need to be analyzed.

Only once analyzed and the threat posed by a given malware sample is estimated a corresponding anti-virus (AV) signature (i.e., a characteristic byte sequence) can be created. As a result, the vast amount of new malware introduces the need for automation of the entire malware estimation process (i.e., collection and analysis).

Advanced large-scale malware collection and analysis infrastructures, such as [2][10][15], can satisfy the requirements for automated tracking of malware, but suffer from several limitations:

1) Despite being executed within an isolated environment, samples must be supplied with requested network services during analysis. Otherwise, the ability to achieve high-quality results is impeded, potentially causing different malware behavior, refusal of execution or even blacklisting of the collector's address space.
   While certain services can be offered using sinkholing techniques, existing approaches are purely network-based, and reactions to malware-initiated connection attempts remain static during runtime. Predefined commonly-used services are offered by these infrastructures to the malware; however, other requests can not be handled accordingly.
2) High-interaction (HI) honeypots pose, aside from their complexity and maintenance issues, high operational risks. These is often inadequately addressed. While there are many methods of mitigation, the remaining risk is still higher than with low-interaction (LI) honeypots, resulting in ethical questions and possibly even legal and liability issues for the operating organization.
3) Existing approaches separate collection and analysis, thus forfeiting the system context (i.e., file handles, requests, sockets) of the victim host. While such separation is not necessarily a limitation (it may not be mandatory to gain qualitative analysis results), we argue that this loss of information hinders analysis and may degrade analysis results or prevent analysis of certain malware altogether.

C. Logic Analysis Using Virtualization

Like modern intrusion detection systems, protocol detection and sinkholing frameworks should not be limited to knowledge about traffic passing through the network. Instead, such systems should incorporate information on the involved systems as well. The tracking of library functions using API-hooks in user or kernel space is one possibility to do so and very popular in malware analysis [27][31][50].

System call tracing has long been known as a solution for profiling and detecting software behavior [17]. While access to privileged operations are usually implemented in the same way on most modern operating systems (OS) (figure 3), not all OSes offer an interface for system call auditing. For operating systems without such an interface, a monitoring functionality has to be implemented by hand, requiring complex and extensive modifications to guests' internal system call handling mechanisms [7].

Hooking of library functions allows finer-grained tracking of a sample's activities than system call tracing. However,

if implemented within the same context of execution as malware is run in, API-hooking is equally susceptible to detection and circumvention by adversaries but requires fewer changes to the operating system's internal mechanisms [7]. Thus, this method of behavior analysis is preferred in most sandboxes [27][31][50].

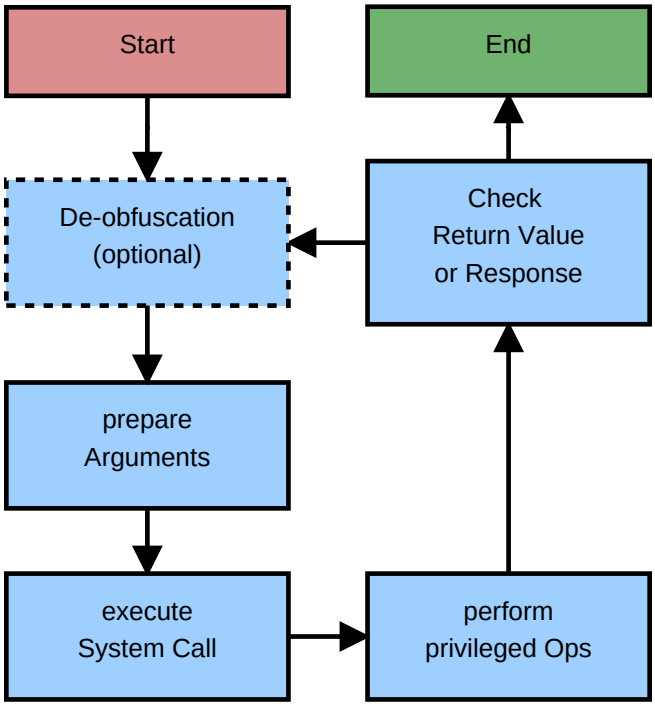

Figure 3. Privileged operations, such as network or disk operations, cannot be executed in user space. Software uses system calls to perform these operations in kernel space.

Due to possibilities offered by virtualization techniques, system call tracing became more popular again. Initially, Tal Garfinkel and Mendel Rosenblum proposed the use of Virtual Machine Introspection (VMI) to extract certain information from a virtual machine (VM) [21]. VMI can also be used to create and enforce behavioral policies [17]. Process activity and the state of a guest's (virtual) hardware components may be used for analysis, too.

We can utilize VMI to extract internal information from a process, monitor and control its behavior. If a program exhibits unexpected behavior or uses system calls or library functions, which are outside of the programs specifications, an intrusion can be assumed.

III. Approach

A. Basic Concept

To identify the services and protocols required in the next step of the execution life cycle of a sample, we intend to harvest information on internal malware logic during execution. In contrast to purely network-based approaches, our method also operates at the binary level, directly interacting with the malware's host system. Therefore, it aims to integrate network-based analysis and binary analysis, as in [48].

As depicted in Figure 4, the presented AWESOME approach [1] is based on an HI honeypot and a virtual machine introspection framework. We enhance our architecture with a transparent pause/resume functionality, which is instrumented to determine and, if needed, interrupt the program flow. Hence, we enable the extraction and alteration of program logic and data within the victim environment during runtime. This is specifically valuable for extracting protocol information and cryptographic material embedded within malware in order to determine the protocol type and intercept encrypted communication.

After checking, extracted information is forwarded to a service handler (SH) and sinkholing service in order to maintain full control over all interactions between the malware and the outside world. For handling unknown traffic as well, finite state machines (FSM) are automatically derived from the observed traffic and used for service emulation. An important goal of automating of the whole collection and analysis process is to handle large amounts of malware while allowing scalability.

B. Added Value

The system context of the malware collection facility persists and is also used in the subsequent analysis. The capabilities resulting from the merge of collection and analysis is similar to the approach used in HI honeypots. Thus, it is more closely aligned to real-world scenarios than LI honeypots. In addition, we achieve increased transparency during analysis due to the use of VMI. We consider this to be a benefit, since we argue that VMI based analysis is more likely to remain undetected by malware.

Compared to other techniques, VMI requires no trusted support components, which could be compromised [14] inside the sample's context of execution. Hence, the approach is more likely to observe the entire malware execution life cycle.

We are able to extract and inject data as well as instructions from or into the memory of the infected virtual machine (VM) during runtime (for example, in order to tap and manipulate encrypted C&C traffic). Since our approach does not depend on analysis components within the VM, we believe it to be more secure while also expecting better overall performance. Moreover, we are able to control any interaction between malware and third party systems. Thus, our architecture can fulfill legal and liability constraints.

Since our approach is applied directly at the instruction level, we are aware of the actions initiated by the malware, thus allowing us to provide matching services and even to service novel communication patterns. Subsequently, the risk resulting from HI honeypot operation is minimized.

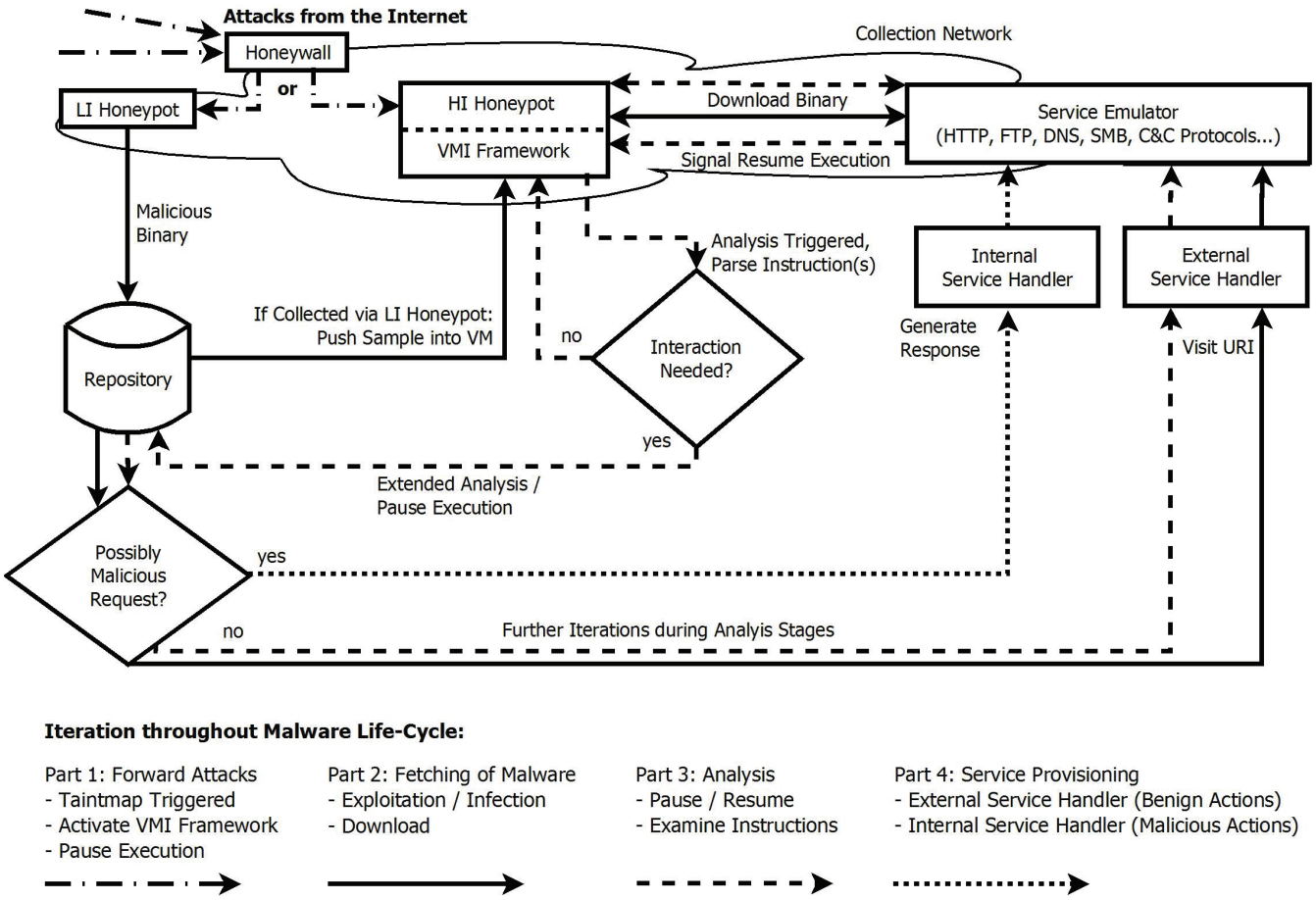

Figure 4. General design of the presented approach.

IV. Design and Implementation

The AWESOME approach [1] utilizes the following components:

- For *malware collection*, a modified ARGOS HI honeypot [41] is used.
- *Malware analysis* is conducted based upon Nitro [38], a KVM-based framework for tracing system calls via VMI. In particular, we determine whether a given action initiated by the currently-analyzed malware requires Internet access and thus apply a complex rule-set to the tracing component.
- Our *service provisioning* component manages all malware-initiated attempts to request Internet resources. Malicious attempts are handled via an appropriate sinkholing service spawned by Honeyd [44], and unknown traffic patterns may be handled utilizing ScriptGen [33].

While most popular LI honeypots have proven to be efficient for malware collection, their knowledge-based approach has also drawbacks regarding the quantity and diversity of the collected malware [51]. With respect to our primary goal (to handle unknown malware), we chose to apply the taint-map-based approach of ARGOS, since it allows the detection of both known and unknown (0-day) attacks. In addition, it is independent of special collection mechanisms. Moreover, it can cooperate with the KVM based VMI framework Nitro. Hence, several components were modified:

1) The victim VM's RTC is detached from the host's clock, since ARGOS is more time-consuming than traditional approaches and thus detectable by an abnormal latency and timing-behavior;
2) Once the taint-map reports tainted memory being executed, we activate the analysis functionality provided by the VMI framework; and,
3) Simple interpretation and filtering of system calls and their parameters is conducted directly within hypervisor space, while more complex analysis is performed via the VMM in the host environment [19].

The entire process consists of three parts (collection, analysis, and service provisioning) and is structured as described below. The steps are repeated iteratively throughout the entire life cycle of the malware.

A. Malware Collection

As a HI honeypot, ARGOS requires much effort in deployment and maintenance. Furthermore, one of the main drawbacks of ARGOS is its poor performance, which is among others related to the overhead caused by the taint mapping technique. To overcome this limitation, we deploy a two stage malware collection network (i.e., a hybrid honeypot system similar to the ones described in [3][28][49]) as outlined in Figure 5.

We take advantage of our existing honeyfarm infrastructure ([4][20]). This malware collection network consists of various honeypots and honeypot-types. Especially, client honeypots play an increasingly important role since the approach of this honeypot type covers the detection of state of the art attack vectors thus enabling client honeypots to capture current malware that may have not been collected using server honeypots.

The honeyfarm utilizes a large-scale network telescope (in particular a /16 darknet) serving the various honeypots. We use this infrastructure in order to filter noise and known malware (in particular everything that can be handled by the LI honeypots or their vulnerability handling modules respectively). The so collected binaries (which we consider to be mostly shellcode containing URLs and droppers) are stored in the central repository. Based on the file-hash known files are distinguished from novel ones. Only novel attempts are forwarded to the ARGOS HI honeypot, which then does the further processing. By doing so we minimize the load on ARGOS and thus justify its operation.

B. Malware Analysis

Dynamic malware analysis utilizing virtualization can be detected and thus evaded by environment-sensitive malware [16][19][34][46]. Hence, our goal is to achieve a reasonably transparent dynamic malware analysis without claiming the approach to be completely stealthy. However, we also consider VMI as the most promising available approach to evade malware's anti-debugging measures due to its minimal footprint. Thus, in order to provide the best chance at evading detection while still gaining the benefits of VMI, we have chosen Nitro since it offers several advantages regarding performance and functionality in comparison to other publicly available tools such as Ether (see [38]).

As Nitro is based on KVM, we have - in addition to guest portability - full virtualization capability, thanks to the host CPU's virtualization extensions. Thus, we can expect reasonable performance.

During the analysis process, we expect a malicious binary to be shellcode or a dropper rather than the actual malware binary. This initially retrieved binary is then decoded and usually contains a URL pointing at the resource used for deploying the next stage of the malware. In the second iteration, execution of this binary continues after it has been downloaded and the VM has been resumed. The resulting system call trace is then examined for routines related to connection handling (e.g., NTConnectPort). If present, we transparently pause execution of the VM and forward related traffic to the service provisioning component. The following sections outline the methods for dynamic malware analysis in more detail.

1) Subroutine Logic Analysis: Based on the executed system call's associated parameters and the memory of the calling process, certain information can be extracted directly from a thread. Traditional semantics checking may prove to be efficient for detecting known subroutines and protocol implementation [42].

Promising candidates for detection are operations and checks performed on the expected peer's response as well as common library functions and routines [50]. A sample's code could be checked for well-known standard algorithms, like routines used to generate symmetric checksums and cryptographic hash functions imported from libraries such as the de-facto standard implementations in OpenSSL or GnuTLS.

If the library function or algorithm being used can be detected, the logical next step is to extract input data. A function should perform sanity checks on its parameters, and on data returned from called functions. Operating systems APIs for use by customers are usually documented; thus, their expected input and return values are known.

Based on this information, the information returned by a function can be analyzed. Such data might be a checksum or signature against which a response was checked or simply a string or sequence of bytes.

The currently decoded part of malware can be analyzed. Sequences of system calls can then be used to reveal more and more deobfuscated parts of malware. Sometimes, it may be easier to not immediately start analysis, but continue execution and wait for a pre-known event to occur. In the case of network communication, analysis may be delayed until malware has sent a packet, and it leaves the virtual machine's (VM) network interface.

2) Delayed Analysis Triggering: Current virtualization solutions try to keep the software layer between physical hardware and the VM as slim as possible in order to improve overall performance. Auxiliary components like network interfaces, however, exist purely in software. To increase the quality of results, subsystems of the VMM and virtual hardware could be used as additional information sources. System call tracing could be used to activate secondary analysis functionality integrated within emulated hardware.

When using hardware assisted virtualization, certain physical devices can be forwarded to a guest exclusively. Ignoring this feature, the emulation code for virtual network interfaces (VIFs) can be extended to hold or trigger analysis components. In the case of qemu, code related to delayed analysis checking could reside, for instance in the virtual network interface (nic.c), as shown in Figure 6.

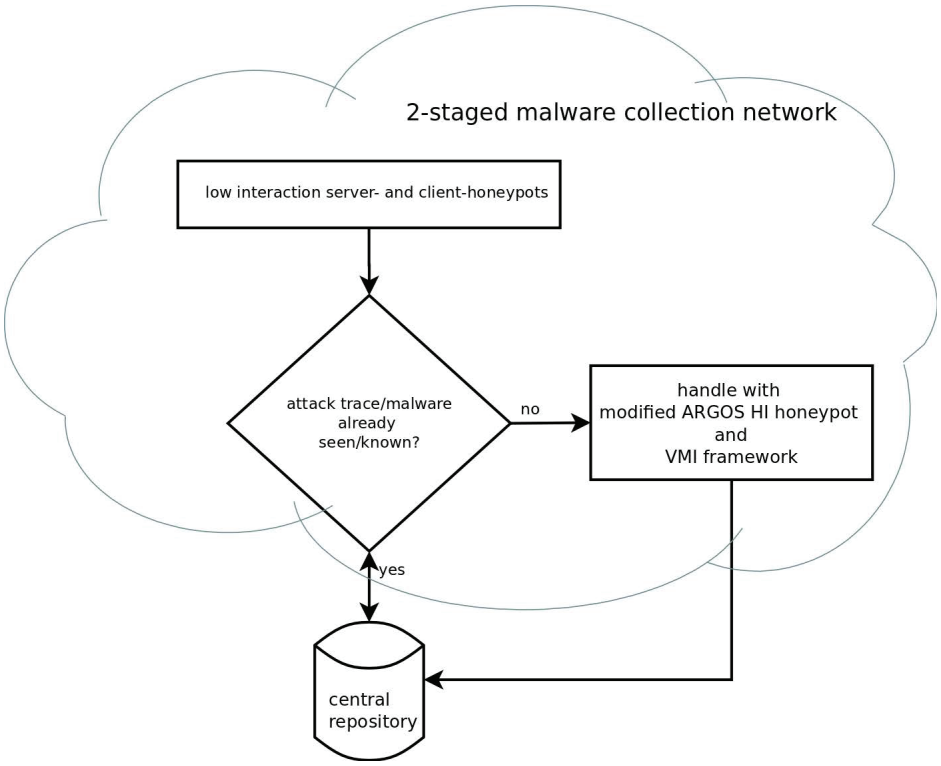

Figure 5. Scheme of the two-stage malware collection network.

Compared to immediate analysis upon execution of a system call, delayed analysis may ensure that malware executes a little bit more of its functionality, finishing execution of its privileged operation (e.g., opening a socket and connecting to a remote host) and returns to ring 3. Malware will normally fall back into a wait state (waiting for a peer's response) after sending a packet, resulting in more code being deobfuscated.

3) System Call Sequence Based Behavior Identification: The approach used for tracing system calls implies that evaluation is possible not just for a single system call and its associated memory, but for series of system calls. As proposed in [17][25][29][43], the behavior of a program or its deviation from its standard behavior can be detected based on series of system calls.

As only relatively short runs of system calls are needed for profiling algorithms, it is possible to detect segments of code instead of complete applications. Behavior detection can also be used to identify state machines present in malware, if context information is examined during analysis. In conjunction with fuzzing techniques, the state machine used in a command and control protocol could also be explored this way.

4) Latency and a Virtual Machine's Real Time Clock: Together, all these analysis steps and operations require a great deal of processing time and a rather sophisticated analysis subsystem. Complex parsing can be done within the hypervisor; however, this would slow down the whole system considerably. As such, the hypervisor should be kept as slim as possible and trigger functionality located within the userland part of the VMM [38].

While being trapped inside hypervisor, a system call and the VM issuing it will remain stalled. Once analysis continues outside the hypervisor, the VM will resume execution. The guest system should remain paused while it is accessed by external analysis components, to keep the guest in a consistent state. The VM should preferably not be able to detect that it is halted; it should retain its internal clock and run detached from the host's real-time clock.

C. Service Provisioning

Malware-driven outbound requests are evaluated to prevent harm to third party systems. For these checks, we rely upon existing measures, such as IDSs or a web application firewall. We are aware that such measures will not be sufficient to tell benign and malicious flows apart in every case; thus, we may build on existing approaches. such as [32]. We assume that a purely passive request (e.g., a download) does not cause harm to a third party. It is thus considered to be benign and handed over to the *external service handler* (SH, see Figure 4).

Since the external SH has Internet access, it resides in a dedicated network segment separated from the analysis environment. If a given request cannot be determined to

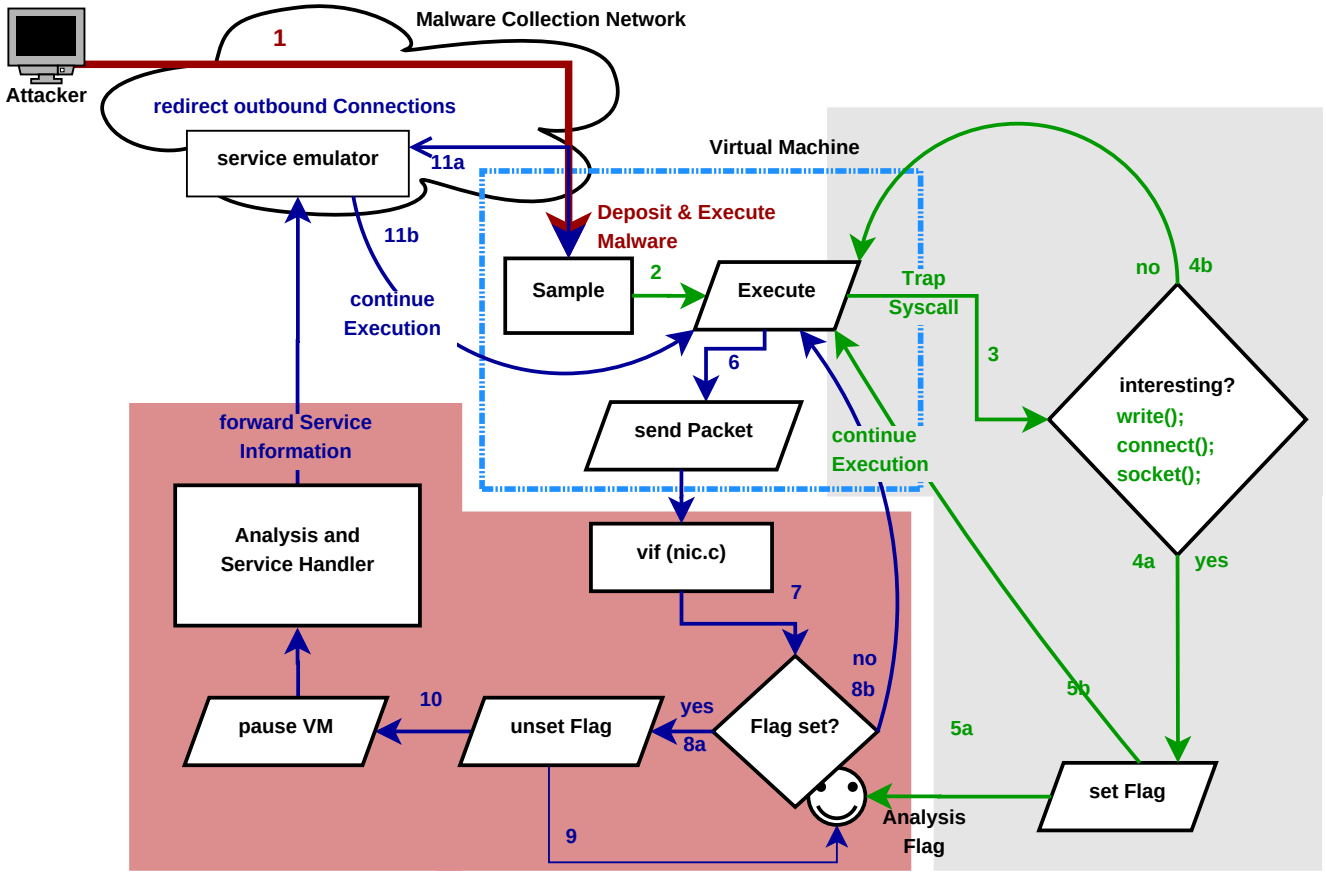

Figure 6. The gray area contains the system call tracing logic and will set a flag if a matching system call has been encountered. The red area holds the analysis components, which are activated only if the flag has been set.

be benign, it is redirected to the *internal service handler*. The sole task of these SHs is to fetch, prepare and provide information for the *service emulator* (SE). The SE launches the requested service in order to deliver the appropriate payload supplied by the SH.

Afterwards, execution is transparently resumed. Since these services can be extremely heterogeneous, the SE is based on Honeyd. It is a very flexible and scalable tool, which is able to emulate or spawn arbitrary services, given that a protocol template exists.

The creation of templates for novel protocols is a much more challenging task. Therefore, we instrument *ScriptGen*, which can derive FSMs from observed traffic, however, it could be replaced by other tools implementing similar approaches [8][12][35].

Each FSM represents the behavior of a given protocol at an abstract level while not depending on prior knowledge or protocol semantics. Based on the generated FSMs, service emulation scripts for the SE can be derived. By integrating such a tool into our approach, we aim toward adding 'self-learning capabilities' to the service provisioning element. Obviously this requires (at least) one-time observation of a given communication between the honeypot and the external system. Hence we need a (supervised) back-channel for learning about novel protocols. Once a corresponding communication has been recorded and the appropriate FSM has been generated, we are able to handle the new protocol as well. While the need for a back-channel is a clear limitation, we consider it to be a reasonable trade-off. The following sections describe the techniques for traffic redirection in more detail.

1) Connection Redirection Techniques: When packets leave a collection or analysis system and the protocol being used has been identified, the respective packet or connection should be forwarded to an appropriate SH within the analysis environment. The SH can either be hosted remotely in the analysis and collection network or locally. Depending upon the approach employed for trapping a sample's actions, redirection of generated traffic can happen in different ways, as shown in Figure 7.

DNS can be used to redirect a connection, if malware relies on using the domain name system to resolve the IP of its peer. The traditional approach is to redirect connections on a per-packet level using network address translation

(NAT), as not all malware will rely on DNS. Network based approaches deploy packet rewriting either on the local virtualization host, or the gateway. In a honeyfarm, the honeywall [11] would redirect these connections as they pass through.

2) Socket Modification at a Binary Level during Runtime: A novel approach to traffic redirection is to rewrite a newly-created socket upon creation in memory, as can be seen in Listing 1. Based on specific system calls, analysis components cannot just extract the parameters used for setting up a new socket or connection. As manipulation of these parameters is possible at runtime, these can be replaced, assuming the location and the format of the expected parameters is known.

While this approach might be more complex than the traditional network-based approach, it offers some interesting advantages, depending on where the destination related data is replaced or manipulated. Using this approach, *IPSec authentication header* functionality might be bypassed relatively easily. Therefore, the SH could use a standard *IPSec* implementation without the need to modify or disable AH integrity checking.

Listing 1. A network related function, for instance the connect command as shown here, could upon execution be rewritten.

```
Trapped API Call:
CALL, cr3: 0xe936000 pid: 1672,
CONNECT(ip: 184.170.X.Y, port: 3127);

Executed API Call:
CALL, cr3: 0xe936000 pid: 1672,
CONNECT(ip: 10.0.1.254, port: 3127);
```

Redirecting a socket at the binary level also reduces side effects occurring due to in-depth analysis. For instance, network-based connection redirection, classification and logging may lead to changed timing behavior; this change of behavior could be detected by malware, lead to connections timing out, or cause certain protocols to stop functioning. Socket redirection at the binary level also allows local system information to be changed more easily.

Naturally, protocol redirection by rewriting socket parameters at runtime is only possible, as long as malware used the operating systems interfaces. If malware would circumvent the operating system, execute completely in kernel mode, or run on a higher privilege level than the kernel, this would no longer be possible. As malware first has to gain the access to these restricted locations sections through system calls, the attempt to do so could in turn be detected. Additional research should be done in this area to investigate further advantages.

As described in section IV-B2, system calls can also be used for initiating delayed analysis functionality in subcomponents of the virtual machine. As network traffic passing through VIFs can be evaluated and manipulated, connection redirection could also be implemented in this location.

All further network related operations should subsequently be performed and handled by the relevant SH, as described in the next subsection.

D. Protocol Sinkholing

For sinkholing several advanced approaches exist on which we can base on. For example *Truman*[1] *(The Reusable Unknown Malware Analysis Net)* and *INetSim (Internet Services Simulation Suite)* simulate various services that malware is expected to frequently interact with. To this end, common protocols, such as HTTP(s), SMTP(s), POP3(s), DNS, FTP(s), TFTP, IRC, NTP, Time and Echo are supported. In addition, INetSim provides dummy TCP/UDP services, which handle connections at unknown or arbitrary ports. Hence these approaches can interact with a given malware sample to a certain level.

Trumanbox [22] enhances the state of affairs by transparently redirecting connection attempts to generic emulated services. To this end, it uses different information gathering techniques and implements four different modes of operation, which allow the application of different policies for outgoing traffic. Thus, Trumanbox can provide different qualities of emulation and addresses issues in protocol identification, transparent redirection, payload modification and connection proxying.

In addition, several work has been conducted in the context of malware analysis to address the issues of detecting, observing and intercepting malicious (C&C-) traffic [10][39][23][45] resulting in publicly available tools.

Hence we concentrate on the issue of handling unknown traffic patterns, such as C&C protocols, within our service provisioning element by instrumenting ScriptGen [33] .

In order to ensure defined test conditions, we chose to build our own malware, since this provides full control over all test parameters thus assuring reproducibility [5]. To this end, we base upon the source code of the Agobot / Phatbot family and compile it using a custom configuration. We chose Agobot, since it is one of the best known bot families and widely used. In addition, it provides a variety of functions. For the sake of easiness, we use unencrypted IRC as the C&C protocol. We deploy a minimal botnet consisting of only one infected host and one C&C server. In addition, we use a third host, which is responsible for the service emulation part. Our resulting test setup consists of three distinct machines:

1) The *victim host* resides on a machine running Microsoft Windows XP SP2. Traffic is captured on this host using WinDump v3.9.5 (the Windows port of tcpdump) based on WinPcap v4.1.2. This host is infected with our malware in order to capture the

[1] http://www.secureworks.com/research/tools/truman/ 10.06.2013

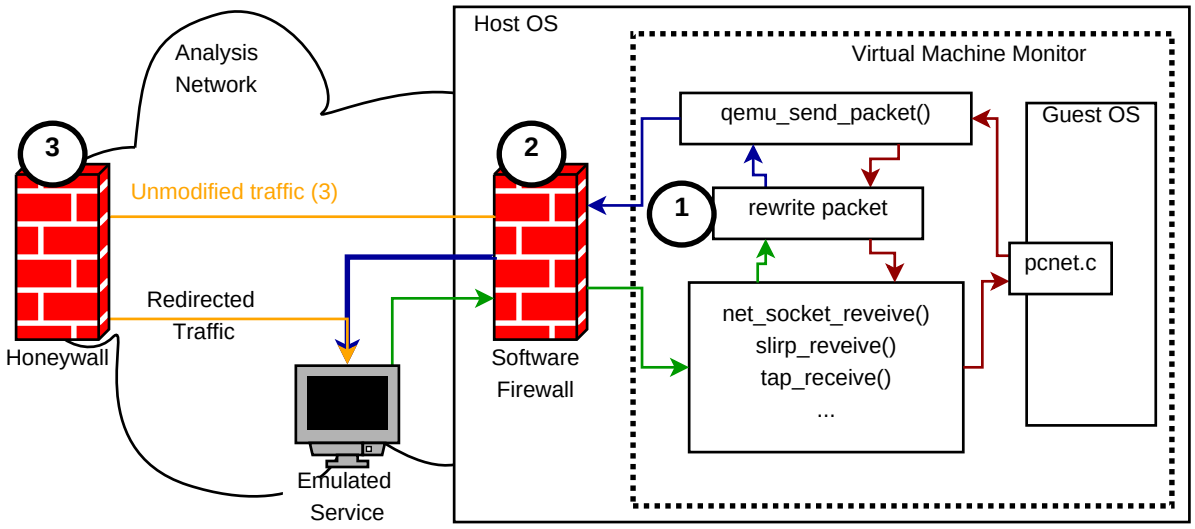

Figure 7. A connection could be redirected within virtual hardware, or the system call itself could be rewritten(1). The packets could be redirected using NAT, either by the virtualization host(2) or the honeywall(3).

malware initiated network traffic as a reference for real C&C traffic.

2) The *C&C server* resides on a machine running Microsoft Windows XP SP3 and is updated to the latest patch level. This is necessary, since this host is also used to build our customized version of Agobot. For a successful build of Agobot, a full featured Microsoft Visual C++ environment including the latest Visual Studio service pack and the latest platform SDK is required. The main purpose of this host is to act as the C&C server. Therefore, we use UnrealIRCd, a well-known IRC daemon widely used by botmasters. There are several customized versions, which are optimized for botnet usage (e.g., designed to serve a vast number of bots). However, for our simple test case the latest standard version (UnrealIRCd 3.2.9) is sufficient.
3) The *service emulator* runs a standard installation of Ubuntu Server 11.10 32bit. Its main task is to process the recorded traffic dump using ScriptGen and to generate a service emulation script out of the resulting FSM, which is then used by Honeyd. To this end, this host is equipped with Honeyd and the dependencies of ScriptGen (i.e., python 2.7, python-dev, cython, python-numpy, python-pcapy, python-setuptools and the nwalign package). Finally, this machine replaces the original C&C server by running Honeyd with the previously created script.

The overall test procedure of our experiment consists of the following steps, which are described in the sections below.

First, we deploy our botnet using a real IRC daemon as C&C server. We launch several commands to generate and record distinct traffic patterns between the infected machine and the C&C server.

Second, we derive FSMs out of this recorded traffic, which are then used to generate a Honeyd service emulation script incorporating the abstract protocol behavior.

Finally, we replace the original C&C server with the service emulation host running Honeyd and the generated script. We launch different commands to evaluate, whether the created script can emulate sufficient responses to fool our bot.

1) Generation of C&C Traffic: First, on the C&C host, we build our custom version of Agobot. Beside some other basic settings, we instruct the bot to connect to our C&C server and join a channel using the respective login credentials and a randomly generated nick. Thereby the nick consists of a random combination of letters prefixed by the string ”bot-” in order to generate data, which is variable on the one hand, but has a significant meaning on the other hand. In addition, we build the bot in debug mode in order to be able to track its activities. The IRC daemon is setup accordingly.

Listing 2. An excerpt of a FSM-recorded conversation.

```
=== Item 201
{ <CONV TCP 80
src:('192.168.1.1', 64459)
dst:('192.168.1.2', 80)>
        [MSG d:I 1:127]
        [MSG d:O 1:49]
        [MSG d:I 1:178]
        [MSG d:O 1:39]
        [MSG d:I 1:262]
        [MSG d:O 1:39 f:Cc]
}
```

Listing 3. Part of the definition of a FSM

```
...
<S [TCP:31337:] f:6 kids:1 label_len:158 new:3 >
{ <TR REG f:6,\#r:1> |F|
 <S [TCP:31337:1] f:2 kids:1 label_len:6 new:4 >
 { <TR REG f:2,\#r:13> |F||Mf||F||Mf||F||Mf||F||Mf||F||Mf||F|
  <S [TCP:31337:1|1] f:2 kids:1 label_len:16 new:0 >
  { <TR NULL f:2>
   <S [TCP:31337:1|1|1] f:0 kids:0 label_len:1054 new:2 >
    {...
```

Next, we execute our customized Agobot on the victim host, which connects back to the configured C&C server and attempts to enter the programmed channel awaiting further commands. On the victim host we record the traffic. Thereby we set additional parameters to avoid a limitation of the recorded packet size, since important information may be truncated otherwise. In order to command the bot we launch a conventional IRC Client, connect as botmaster to our C&C Server and join the previously configured channel as well. We instruct the bot to execute a given command by performing a query. Thereby we use a full stop as command prefix.

After login to the bot (*.login User password*) we instruct it to perform some harmless actions, such as displaying status information (*.bot.about, .bot.sysinfo, .bot.id, .bot.status*). Thereby we launch a diverse set of commands in order to obtain representative data. In particular, we use specific commands, such as .bot.sysinfo, several times, since they also query random and regularly changing values (e.g., uptime).

Furthermore, we also send some dummy commands and random strings intended to insert ”noise”. This is afterwards used to examine the result, i.e., none of these commands should appear in the resulting script. We perform a number of such sessions using randomly chosen commands in an arbitrary order. In doing so, we simulate a number of distinct bots, since the bot generates a different nick for every session. Finally, the recorded traffic of these conversations is filtered to remove traffic produced by other applications running in background.

2) Traffic Dissection and FSM Generation: On the service emulation host we dissect the previously recorded traffic dump and extract the used ports within the communication. Since ScriptGen is port based, this analysis is necessary to determine for which ports a corresponding FSM needs to be generated. As a result, we receive a list of identified ports and generate a FSM for each port. Therefore, we apply the existing functions implemented by ScriptGen:

First, a simplified FSM is built by parsing the traffic dump file for the corresponding data-link type and reassembling the packets and the respective conversations. A conversation is composed of messages, whereas a message is the longest set of bytes going in the same direction. Thus it is the starting point to build the FSM. The rebuild of the resulting conversations is based on the unique tuple of source- and destination-address and the corresponding ports, as can be seen in Listing 2. Thus there is no check, whether a given port actually corresponds to the expected protocol but one FSM per port is built.

It outlines the mentioned unique tuple (source- / destination-address, source-/destination port) along with the messages, where
d is the direction from the server perspective (I: incoming, O: outgoing),
l is the length of the payload in bytes and
f describes the set flags (if any). In this example ”Cc” refers to ”client close”.

Next, functionality to attach data contained in the traffic dump to an eventually existing FSM are called. In addition, these functions could be used to infer content dependencies between known conversations and an existing FSM. The output is a serialized, updated FSM (see Listing 3) serving as input to our converter, which implements the Region Analysis and builds the actual FSM based on the chosen thresholds for macroclustering and microclustering.

It contains information about the used protocol, the observed states and edges as well as the respective transitions, where
S is the self-identifier (i.e., the protocol and port) followed by the path,
f is frequency of the state,
kids describes the number of transitions the state has,
label_len is the length of the state labels,
new is the amount of conversations and
TR (Type Region) describes the identified region type. A region can thereby be *NULL* describing a transient state, i.e., a state with an outgoing NULL transition. That is, a state that immediately leads to a new future state after label generation without expecting a client request.

In addition, a region can be identified as *Fixed* (containing repeatedly the same data), *Mutating* (containing varying

data) or as *REG* (i.e., a regular expression). In turn, a region described via a regular expression may consist of several regions having varying characteristics. This is optionally indicated via corresponding flags, such as *"F"* (fixed region) or *"Mf"* (mutating region).

Listing 4. For a given path the corresponding strings and regular expressions representing the incoming and outgoing messages as identified by ScriptGen are extracted.

```
<<< Incoming
> Fixed region
0000    50  41  53  53  20  31  32  33          PASS 123
0008    0D  0A                                  ..
%regexp: 'PASS\\ 123\\\r\\\n'
<<< Incoming
> Fixed region
0000    4E  49  43  4B  20  62  6F  74          NICK bot
0008    2D                                      -
> Mutating region
Content candidate
...
> Fixed region
0000    77                                      w
Content candidate
...
> Fixed region
0000    0D  0A  55  53  45  52  20  62          ..USER b
0008    6F  74  2D                              ot-
...
>>> Outgoing (len: 16)
0000    50  49  4E  47  20  3A  35  44          PING :5D
0008    38  42  34  37  34  34   D   A          8B4744..
0010
**********************
```

Finally, every generated FSM is inspected and basic information such as port, IP address and the identified protocol is extracted for further processing (e.g., a rule-based decision, whether the FSM for the respective port should be enabled in the service emulator). We incorporate the described functionality for traffic dissection and FSM generation into a small Python script.

3) FSM Traversal and Script Building: In order to generate a service emulation script out of a given FSM, information about its elements is required. Specifically, we need to determine the total number of states and for each state

- all state labels (the messages sent by the server),
- the total number of edges (the number of possible transitions towards the next state) and for each edge
  - the respective edge label (the message representing the client request triggering the transition)

Thereby we assume that the first seen state label represents the initial message sent by the server (e.g., a service banner). Thus it is defined as the default state. To this end, a certain FSM is inspected by traversing through its paths, which are derived from the identified regions, as depicted in Listing 4. Based on this information the sequence of regular expressions and the respective strings to respond with is reassembled. We find, that we can recognize our previously exchanged messages, i.e., the launched commands and their according responses. In particular, we conclude, that ScriptGen is able to map also variable values correctly. For instance, it identifies *"USER bot-"* and *"NICK bot-"* as fixed regions while defining single letters and their combinations as mutating regions. This matches our configuration instructing the bot to use a variable nick consisting of random letters prefixed by "bot-". Thus, ScriptGen abstracts the protocol semantics correctly in this example. The output of the FSM traversal is then included in the service emulation script file. To this end, the static part of the script is created in a first step consisting of a header and some basic functions for echoing fake messages. For basic data exchange we rely upon basic functionality of Honeyd. In a second step, we created a modified version of the traverse function as originally implemented in ScriptGen so that it now traverses through a given FSM path and extracts the regular expressions of the exchanged messages.

Listing 5. A (simplified, primitive and reduced) FSM generated by AWESOME.

```
# State S0)
S0)
  if [[ $_ =~ /'PASS \\$PASS\\\r\\\n' ]];
  then
    echo -e "OK"
  else
    echo -e "ERROR"
    state="S2"
  fi
;;
# State S2:
S2)
  if [[ $_ =~ /'NICK\\ bot\\-untuv\\\r\\\nUSER
       \\ bot\\-untuv\\ 0\\ 0\\ \\:bot\\-un' ]];
  then
    echo -e "^@^@^@^@^@^@"
  else
    echo -e "ERROR"
    state="S3"
  fi
;;
# State S3:
# transient state (immediately leads to a new
# future state without expecting a client request)
S3)
  # therefore no response is generated
  # -> traverse directly to S4
  state="S4"
;;
... omitted
# State S9:
S9)
  if [[ $_ =~ /'USERHOST\\ bot\\-untuv\\\r\\\n' ]];
  then
    echo -e ":$IP 302 bot-untuv :bot-untuv=+"
  else
    echo -e "ERROR"
    state="S10"
  fi
;;
# State S10:
S10)
  if [[ $_ =~ /'JOIN\\ \\#$CHANNEL\\#\\ $PASS\\\r\\\n' ]];
  then
    echo -e ":$IP 302 bot-untuv :bot-untuv=+"
  else
    echo -e "ERROR"
    state="S11"
  fi
;;
```

4) Results: As a result of the steps described above a service emulation script is generated and can be used with Honeyd. Thereby the regular expressions intended to cover

a given class of requests are of special interest.

After replacing the original C&C server with the service emulation host we launched different commands to evaluate, whether the created script can emulate sufficient responses. By interacting with the emulated IRC service we found that it is capable of generating appropriate responses to a set of very basic requests. However, slight deviations of these basic requests cause the emulated service to not respond at all thus leading to insufficient emulation. We believe, that this is related to the limited amount of traffic, that we have generated for this experiment. In fact, the size of the generated traffic dump file is less than 100kB, which seems clearly insufficient for ScriptGen to learn all interactions properly. While we were able to demonstrate that generating service emulation scripts using the ScriptGen approach is essentially possible, further experimentation will be necessary to produce more accurate results.

Further experimentation is necessary and we are confident, that a larger amount of traffic will produce more accurate results. However, we found that generating service emulation scripts using the ScriptGen approach is essentially possible, an example is depicted in Listing 5. Specifically, we believe, that the basic assumption of ScriptGen (i.e., an exploit performs a limited number of execution paths) can be applied to our use case of service emulation for C&C traffic, since a bot performs a limited number of commands as well. Thus we conclude that the application of ScriptGen within the service provisioning component of our presented approach is feasible.

V. Discussion and Future Work

From a conceptional perspective the main limitation of our approach is that it can only capture samples of autonomous spreading malware due to the use of a taintmap. The server based approach with taintmaps, i.e., passively waiting for incoming exploitation attempts, implies that this type of honeypot can not collect malware, which propagates via other propagation vectors such as Spam messages or drive-by downloads.

With respect to the outlined paradigm shift in attack vectors we will need to consider such propagation vectors as well in future. However, since this is a sensor issue, this limitation may be overcome by integrating corresponding honeypot types (i.e., client honeypots) into our approach. This is left for future work. Moreover, due to the ongoing spread of IP-enabled networks to other areas (e.g., mobile devices and SCADA environments) our approach will be required to integrate malware sensors covering these attack vectors as well in future.

Beside the necessary further experiments to improve the accuracy of service emulation for C&C traffic, enhancements in malware analysis need to be tested. In particular, the interaction between all stated components will be evaluated, once all of them are readily deployed. At the time of writing the evaluation of system calls produced by Nitro needs to be finished and measures for checking malware initiated outbound attempts need to be evaluated. In a next step, we will test the use case of tracking and intercepting encrypted C&C protocols as well as making use of malware calling library functionality to perform subroutine analysis.

When it comes to malware analysis itself, multiple paths of execution could be traversed. The state of a virtualized guest could be saved at multiple times during execution and, later on, the analysis environment could revert the guest to different states saved during processing. Traversal of multiple paths of execution would allow automated fuzzing to take place during analysis. Multiple possibly valid responses may be tried, if few candidates exist. Additionally, unknown state machine versions used for C&C traffic may be explored.

Future work will also include the completion and evaluation of the service emulator and the measures to prevent harm to third party systems.

VI. Conclusions

In this paper, we have presented a novel approach for integrated honeypot-based malware collection and analysis, which extends existing functionalities. Specifically, it addresses the separation of collection and analysis, the limitations of service emulation, and the operational risk of HI honeypots.

The key contribution of the approach is the design of the framework as well as the integration and extension of the stated tools. While this is an ongoing research activity and thus still under development, several modifications to ARGOS and Nitro have already been implemented and successfully tested, indicating the feasibility of our approach.

System call tracing alone is a rather finite source of information for malware analysis, as relatively little information worth analyzing is transferred between a system call routine and the caller. System call parameters, such as path names or URIs, are extremely valuable in certain situations, but to make full use of all available information, the virtual machine's memory has to be analyzed too. Static analysis tools can subsequently process deciphered binary samples in memory and no longer have to deal with unpackers or loaders, and thus handle polymorphic and metamorphic software with relative ease.

By tracking a binary using a series of system calls, more and more pieces of malware can be revealed and evaluated. This results in an outcome very much desired by malware researchers; the longer malware operates, the more information can be extracted from malware with less overhead for dealing with obfuscation [13][36][38].

Cryptographic key material and parameters used for establishing and maintaining a secure tunnel between peers would be extremely useful in protocol emulation. These values can be extracted or even replaced within the guest's memory. While replacing such data automatically at runtime may be

challenging, it would allow emulation of an encrypted peer using valid and trusted key material.

While the service provisioning element raises several technical issues regarding protocol identification, connection proxying, transparent redirection, payload modification as well as detection, observation and interception of malicious (C&C-) traffic, most of these issues can be addressed based on existing work. We referred to related research in this area and focused on the issue of handling unknown traffic patterns, such as C&C protocols.

To this end, we presented a proof of concept implementation leveraging the output of ScriptGen for use within the service provisioning component of our presented approach. Using this proof of concept, we evaluated the feasibility of using ScriptGen for generating service emulation scripts intended to spawn an emulated C&C service. We setup a minimal botnet using customized malware in order to generate the corresponding C&C traffic. Out of this recorded traffic we derived FSMs, which were then used to generate a service emulation script incorporating the abstract protocol behavior.

REFERENCES

[1] Martin Brunner, Christian M. Fuchs, and Sascha Todt. Awesome - automated web emulation for secure operation of a malware-analysis environment. In *Proceedings of the Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2012)*, pages 68–71, Rome, Italy, August 2012. International Academy, Research, and Industry Association (IARIA), XPS. ISBN: 978-1-61208-209-7. Best Paper Award.

[2] M. Apel, J. Biskup, U. Flegel, and M. Meier. Early warning system on a national level - project amsel. In *Proceedings of the European Workshop on Internet Early Warning and Network Intelligence (EWNI 2010)*, January 2010.

[3] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and N. Provos. A hybrid honeypot architecture for scalable network monitoring. 2006.

[4] M. Brunner, M. Epah, H. Hofinger, C. Roblee, P. Schoo, and S. Todt. The fraunhofer aisec malware analysis laboratory - establishing a secured, honeynet-based cyber threat analysis and research environment. Technical report, Fraunhofer AISEC, September 2010.

[5] Martin Brunner. Integrated honeypot based malware collection and analysis. Master's thesis, 2012.

[6] BSI. Die lage der it-sicherheit in deutschland 2011. Bundesamt fuer Sicherheit in der Informationstechnik, May 2011.

[7] David M. Buches. Fast system call hooking on x86-64 bit windows xp platforms, April 2010.

[8] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM.

[9] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: the commoditization of malware distribution. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[10] D. Cavalca and E. Goldoni. Hive: an open infrastructure for malware collection and analysis. In *proceedings of the 1st workshop on open source software for computer and network forensics*, 2008.

[11] Jay Chen, John McCullough, and Alex C. Snoeren. Universal honeyfarm containment. Technical Report CS2007-0902, New York University and University of California, San Diego, 9500 Gilman Dr., La Jolla, CA 92093, USA, September 2007.

[12] W. Cui, V. Paxson, Nicholas C. Weaver, and Y H. Katz. Protocol-independent adaptive replay of application dialog. In *In The 13th Annual Network and Distributed System Security Symposium (NDSS*, 2006.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[14] M. Dornseif, T. Holz, and C.N. Klein. Nosebreak - attacking honeynets. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, june 2004.

[15] M. Engelberth, F. Freiling, J. Göbel, C. Gorecki, T. Holz, R. Hund, P. Trinius, and C. Willems. The inmas approach. In *1st European Workshop on Internet Early Warning and Network Intelligence (EWNI)*, 2010.

[16] P. Ferrie. Attacks on virtual machine emulators. In *AVAR Conference, Auckland*. Symantec Advanced Threat Research, December 2006.

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128, 1996.

[18] Jason Franklin, Adrian Perrig, Vern Paxson, and Stefan Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 375–388, New York, NY, USA, 2007. ACM.

[19] Christian M. Fuchs. Deployment of binary level protocol identification for malware analysis and collection environments. Bacherlor's thesis, Upper Austria University of Applied Sciences Hagenberg, May 2011.

[20] Christian M. Fuchs. Designing a secure malware collection and analysis environment for industrial use. Bacherlor's thesis, Upper Austria University of Applied Sciences, Bacherlor's degree programme Secure Information Systems in Hagenberg, January 2011.

[21] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[22] Christian Gorecki. Trumanbox - improving malware analysis by simulating the internet. RWTH Aachen, Department of Computer Science, 2007. Diploma Thesis.

[23] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *NDSS*. The Internet Society, 2008.

[24] P. Gutmann. The commercial malware industry. In *DEFCON 15*, 2007.

[25] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[26] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. Technical report, University of Mannheim, Laboratory for Dependable System, 2008.

[27] Honeynet Project. Know your enemy: Sebek, November 2003. last visited: 2011-03-11.

[28] X. Jiang and D. Xu. Collapsar: a vm-based architecture for network attack detention center. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association.

[29] Xuxian Jiang and Xinyuan Wang. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 198–218, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Tobias Klein. *Buffer Overflows und Format-String-Schwachstellen: Funktionsweisen, Exploits und Gegenmassnahmen*. Dpunkt.Verlag GmbH, 2004. ISBN 9783898641920.

[31] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. *Security and Privacy, IEEE Symposium on*, 0:29–44, 2010.

[32] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. Gq: practical containment for measuring modern malware systems. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 397–412, New York, NY, USA, 2011. ACM.

[33] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, Washington, DC, USA, 2005. IEEE.

[34] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection (RAID) Symposium*, 2011.

[35] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 110–125, Washington, DC, USA, 2009.

[36] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 441–450, Washington, DC, USA, 2009. IEEE Computer Society.

[37] G. Ollmann. Behind today's crimeware installation lifecycle: How advanced malware morphs to remain stealthy and persistent. Whitepaper, Damballa, 2011.

[38] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*. Springer, November 2011.

[39] Daniel Plohmann, Elmar Gerhards-Padilla, and Felix Leder. Botnets: Detection, measurement, disinfection & defence. European Network and Information Security Agency (ENISA), 2011.

[40] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost turns zombie: exploring the life cycle of web-based malware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, USA, 2008. USENIX Association.

[41] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 15–27, 2006.

[42] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *SIGPLAN Not.*, 42:377–388, January 2007.

[43] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, page 18, Berkeley, CA, USA, 2003. USENIX Association.

[44] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[45] Konrad Rieck, Guido Schwenk, Tobias Limmer, Thorsten Holz, and Pavel Laskov. Botzilla: detecting the "phoning home" of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1978–1984, New York, NY, USA, 2010. ACM.

[46] J. Rutkowska. Red pill... or how to detect vmmusing (almost) one cpu instruction, 2004. http://invisiblethings.org.

[47] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86. In *In Proceedings of CCS 2007*. ACM Press, 2007.

[48] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Gyung Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security.*, 2008.

[49] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM symposium on Operating systems principles*, SOSP '05, New York, NY, USA, 2005. ACM.

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

[51] J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *Proceedings of the 9th international conference on Information and communications security*, ICICS'07, Berlin, Heidelberg, 2007. Springer-Verlag.