# Dark Application Communities


Michael E. Locasto
Network Security Lab
Dept. of Computer Science
Columbia University
locasto@cs.columbia.edu

Angelos Stavrou
Network Security Lab
Dept. of Computer Science
Columbia University
angel@cs.columbia.edu

Angelos D. Keromytis
Network Security Lab
Dept. of Computer Science
Columbia University
angelos@cs.columbia.edu



## ABSTRACT

In considering new security paradigms, it is often worthwhile to anticipate the direction and nature of future attack paradigms. We identify a class of attacks based on the idea of a "Dark" Application Community (DAC) – a collection of bots and zombie machines that actively performs binary-level supervision of applications to help an attacker automate the process of finding vulnerabilities. A collection of such hosts can observe and attempt to influence the behavior of automatic defense systems. An attacker can use the DAC as both a test platform for subverting security applications and as a reconnaissance network for exploiting commonly deployed automatic update and early warning systems.

An instance of this type of Application Community can host what we call an *automorphic worm*. An automorphic worm is application-agnostic and vulnerability-generic. Such a worm attempts to remain stealthy by cycling through the portfolio of vulnerabilities that the DAC has identified. We examine the underlying principles of a DAC, which are based on the existing paradigm of using security tools to help violate security.



## Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software; I.2.6 [**Learning**]: Knowledge acquisition

## General Terms

Network Security, Application Communities

## Keywords

automorphic, dark application communities


## 1. INTRODUCTION

Botnets have emerged as a major source of problems for systems security, ranging from supporting Distributed Denial of Service (DDoS) attacks against Web services [31, 15, 28] to sending spam email and enabling distributed phishing [32, 10]. The increasing size of real-world botnets and the sophistication of bot malware has promoted them to an effective tool for profit-motivated online crime [10]. Traditionally, the use of botnets has been focused on carrying out attacks and reconnaissance. Relatively little attention has been paid to actively using botnets to automatically identify vulnerabilities and generate exploits. A collection of zombie machines engaged in the latter activities can form what we call a "Dark" Application Community (DAC)[1].

Just as researchers have suggested the idea of using Application Communities [18] to leverage the large resources available in a monoculture for defensive operations, so too can the resources of a botnet be leveraged by attackers. They can do so in at least two distinct ways. First, a DAC can host malware that actively performs binary supervision of applications to automate finding software errors – potential vulnerabilities. Second, members of a DAC can observe the behavior of defense systems, test their thresholds and sensitivity to attacks, and attempt to influence their operation. For the attacker, both propositions are justified by an economic argument. Attackers, even if well-funded, can only automate their bug-finding, reconnaissance, and counter-evasion activities to a certain degree. Identifying vulnerabilities and creating exploits based on them is a time-consuming, manual process.

This process is exactly the process of quality assurance that large software vendors try to perform. The only difference is that once a vendor identifies a vulnerability, they fix it and distribute a patch. When a cracker finds a fault or vulnerability, they construct an exploit to attack it. The process of identifying such faults is very much the same for both entities.

Discovering vulnerabilities is only the first part of a successful intrusion. Crafting malware that is stealthy enough to bypass the variety of security systems that are deployed on the target hosts is also critical. Finally, if the constructed exploit is ultimately discovered, the members of a DAC should be able to take advantage of the alerts emitted by any automated early warning systems. Upon noticing alerts related to the current vulnerability being exercised, a DAC can shift operation to another vulnerability or even blind the early warning system with a bogus exploit or disposable



*NSPW 2006*, Sept. 18–21, 2006. Schloss Dagstuhl, Germany.



[1]Note that we do not use the term "dark" in the sense of dark IP address space, and a DAC does not require a honeynet or other dark IP space monitor. Rather, we use "dark" to refer to the malicious nature of this type of Application Community.

vulnerability. For the rest of the paper, we refer to this new type of attack as a "shift-blind" attack.

## 1.1 Motivation for a DAC

Parallelization helps speed up the task of testing an application for bugs and faults. Some vendors may have enough resources to build a virtual or isolated network infrastructure for this type of testing. Certain popular open source projects may have a large enough developer community and user base to both provide meaningful bug feedback and correct fixes. On the other hand, attackers probably do not have the financial resources to construct their own testbed and employ a QA team. However, attackers can use the large collection of bots under their control as a Dark Application Community, rather than leaving them idle.

Part of the task of a DAC is to supervise the execution of all software on the zombie machine and report errors and faults back to the attacker (along with the input causing such events) to direct his or her research activities. Such distributed fault identification helps parallelize the "debugging" operation, since a large population of end-users executes a variety of applications under a number of configuration settings and many different inputs. Many of the defense systems proposed in the current research literature effectively allow the attacker to expose a previously unknown exploit or vulnerability in order to construct a defense. A DAC, on the other hand, delegates the identification of vulnerabilities to the end users. The mission of a DAC is to assemble a portfolio of vulnerabilities. Having such a portfolio implies a variety of different vulnerability categories, with some vulnerabilities (*e.g.*, remote access) being more valuable for propagation, while others may be more valuable for direct economic gain.

A widespread DAC will most likely contain a variety of security applications in addition to the "normal" array of software installed on machines in its population. The presence of security systems provides the opportunity for the DAC to employ these systems as a type of testbed to vet any generated exploits without risking exposure.

## 1.2 Use Cases

A Dark Application Community can be used to support the spread of malware that exploits a new vulnerability from generation to generation. In doing so, a DAC can also attempt to subvert the operation of automatic defense systems. We call such malware an "automorphic worm."

### *1.2.1 Automorphic Worms*

Most known worms (malware that spreads in an automated fashion) are strictly identical from generation to generation. This statement is somewhat true for polymorphic and metamorphic worms as well; even though new instances may be encrypted or otherwise obfuscated, the operation of the worm remains the same (it repeatedly exploits the same underlying vulnerability to infect a host). Past approaches to worm detection are predicated on rate limiting or exploit signature matching. While such detection techniques are adequate for loud, fast-spreading worms, they are not appropriate for slow, stealthy, and polymorphic worms.

Many recent detection [24, 13, 33] and protection [5, 23, 35] mechanisms seek to identify and deter worms without reference to the particular exploit string or input. That is, this body of work attempts to defeat polymorphic worms by identifying the underlying vulnerability [6] rather than a particular exploit string. Such systems are termed *exploit-agnostic* or *vulnerability-specific*, and they have some hope of identifying and stopping polymorphic worms.

Automorphic worms, on the other hand, are designed to exploit new vulnerabilities with each generation so that current mechanisms (aimed at polymorphic detection and prevention) are not as effective as they might otherwise be. An automorphic worm is a worm that is application-agnostic and vulnerability-promiscuous. This type of stealthy worm actively performs binary supervision of applications to automate finding software errors – potential vulnerabilities.

The term "automorphic" means "patterned after oneself," and this description is an apt one. The next generation aims at the same behavior or set of tasks that the parent generation performs, but augments itself by creating and maintaining a portfolio of exploits. However, rather than rely solely on polymorphic techniques (*e.g.,* encryption) to disguise its content in the hopes of avoiding detection, it spreads via a changing set of vulnerabilities. It should be noted that this portfolio is not composed of a strictly static set of exploits like those employed by multi-vector worms (*e.g.,* Nimda). In this way, an automorphic worm can potentially evade vulnerability-specific protection mechanisms.

### *1.2.2 Anti-Security & Malicious Feedback*

Having generated a portfolio of exploits, an attacker can go one step further and apply them to the botted network (DAC), monitoring and controlling the reaction of the security mechanisms on each of the botted hosts. Since the attacker is already in control of these bots, there is no risk of exposure when a new, possibly untested, vulnerability is deployed. A DAC is also diverse enough to host a plethora of security protection mechanisms such as Vigilante [5] or other commercial malware detection systems, thus transforming the DAC to a formidable anti-security testbed, although some research has been done to harden security mechanisms against this type of analysis[34].

This malicious feedback can be used in various ways, and generating stealthier worms is only one of them. Manipulating and subverting automatic defense systems in the DAC can have even more widespread and devastating effects. Indeed, most modern defense systems activate their signature generation algorithms upon detection of a new exploit. This signature information is transmitted to a set of centralized servers where an assessment about the new exploit can be made to allow a fast reaction to new worms. An automorphic worm, since it operates on a lower level, can force the automatic defense systems in the DAC to generate fake or real exploit signatures. These signatures can be transmitted in a coordinated and distributed fashion to "blind" the early worm warning systems. Such an action effectively creates a Distributed Denial of Service attack to "shift" their attention from the real threat.

The same worm early warning systems can be used to notify attackers about new or discovered vulnerabilities since this type of information is transmitted to hosts in the Internet including botnets. To make things worse, these signature and vulnerability announcements are security application specific, giving the attacker knowledge about the detection capabilities of each individual security application. The attacker can either use this information to scan for susceptible systems or to avoid detection by shifting to an undiscovered exploit.

## 1.3 Contributions

This paper makes three primary contributions:

1. a review of the emerging intrusion defense paradigm of vulnerability–specific (rather than exploit–specific) protection mechanisms and an exposition of the *collaborative insecurity* paradigm: collections of machines that form a Dark Application Community.
2. the identification of a new type of malcode that is one result of this new attack paradigm. *Automorphic worms* can potentially defeat vulnerability-specific filters and protection mechanisms.
3. the introduction of a new type of attack, the *shift-blind* attack. This attack can manipulate the signature transmission and early worm warning systems to "blind" the intrusion detection and host based security systems and "shift" their attention to a fake or disposable vulnerability while the real worm spreads unnoticed.

In order to frame the discussion of these contributions, we next turn our attention to related research.

# 2. RELATED WORK

Automorphic worms are related to work on Collaborative Security, virtualization techniques, kleptography, and automatically spreading malware. Kleptography [38] is the use of cryptography to secretly and securely steal information. Cryptosystems can be constructed by an attacker such that it provides cryptographic services, but also encodes a hidden weakness or subliminal channel. The work most closely related to ours is a combination of Application Communities and subversive virtual machines.

## 2.1 Collaborative Security

Organizations often lack the resources to recognize and respond to large scale threats. Enabling them to leverage the resources of their peers by sharing information related to threats and attacks seems like a promising solution to this problem. Collaborative security is the growing trend towards sharing information security resources within and across administrative domains and systems to improve the overall security of the peer group. Three areas of computer security where a collaborative approach are immediately applicable are (*a*) worm detection and notification, (*b*) self-healing software, and (*c*) spam filtering. The reasoning is that a large, distributed network of sensors can achieve knowledge of an attack faster than a single isolated node.

This observation is a widespread one. In particular, for worm detection [20], notification [21], and containment [1] systems, a collaborative approach is mentioned several times in the literature. Systems that seek to generate signatures for worm traffic include Autograph [11], Polygraph [24], and EarlyBird [30]. All three papers refer to signature distribution as a fundamental step in defending against worms.

A study by Moore *et al.* [21] concludes that a worm containment response needs to occur within three minutes. In addition, the participation of nearly all major AS's is required for a containment to be effective. While these requirements are quite challenging, they confirm that foreseeable threats are best addressed by a collaborative approach.

Vigilante [5] is a system motivated by the need to contain Internet worms. To that end, Vigilante supplies a mechanism to detect an exploited vulnerability. A major advantage of this vulnerability-specific approach is that Vigilante can be exploit-agnostic and can potentially be used to defend against polymorphic worms. Vigilante defines an architecture for production and verification of Self-Certifying Alerts (SCA's), a data structure for exchanging information about the discovered vulnerability. Vigilante works by analyzing the control flow path taken by executing injected code.

Collaborative security can also be leveraged for more mundane intrusion detection tasks. DOMINO [37] is a system for correlating intrusion alerts. Lincoln *et al.* examine the problem of privacy–preserving alert sharing for IDS systems [17], one of the challenges proposed in Du and Atallah [8]. Kruegel *et al.* [14] propose a peer-to-peer system that recognizes attacks in a distributed manner. In their system, only a small number of messages needed to be exchanged to determine that an attack was underway.

A collaborative approach to security also seems useful in the context of self-healing software. Not only can networks and end-hosts exchange information about intrusion alerts, but they can also exchange information about exploited vulnerabilities and code patches for these vulnerabilities. Application Communities [18] are one particular expression of this idea whereby a large collection of hosts agree to collaboratively monitor small slices of each instance of an application locally. Taken together, all peers provide global coverage of the application, even though they execute it independently. When a fault or vulnerability is discovered, information that enables each host to prevent further occurrences of that fault is exchanged with peers.

## 2.2 Vulnerability-Specific Protection

The first attempts at automatically identifying worms and creating signatures from this detection process focused on using certain worm traffic characteristics: spreading and contact rate, uniformity of packet header fields, *etc.* While this type of detection helps with simple and fast spreading worms, it does not help in the case of polymorphic or slow and stealthy worms. Subsequent attempts examined the actual packet content to differentiate executable malcode from normal traffic. Such content-based approaches (*e.g.,* PayL [36], APE [33], Polygraph [24] *etc.*) may work against slow and stealthy worms, but not all polymorphic ones. Current approaches such as Vigilante [5] or VSEF [23] attempt to instrument the host to automatically identify vulnerabilities and then block input that exercises that vulnerable state without reference to a particular exploit string or input, as first proposed by the Shield system [35] for *known* vulnerabilities. Other work attempts to automatically reconstruct a worm's control flow from the captured binary code [13] [4]. Crandall et al. [6] discuss the problem of generating quality vulnerability-specific signatures via an empirical study of the behavior of poly- and meta-morphic malcode. They outline the difficulty of identifying enough features of an exploit to generalize about a specific vulnerability. Focusing on the behavior of malcode seems to be a more promising approach. Some work has been done to generate anomaly-based signatures for web servers [29].

## 2.3 Virtualization

Virtualization is not a new idea; it was first popularly realized in the IBM System/360, but fell into disuse during the advent of personal computers. Recently, the use of virtual machines has come back into fashion in both research and industry to leverage underutilized hardware, reduce management complexity, and provide isolation. The ability to isolate execution contexts and intercept their actions is an attractive capability for security systems.

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research. Virtual machine monitors (VMM) are employed in a number of security contexts from autonomic patching to intrusion detection [9]. King *et al.* [12] propose implementing rootkits with binary supervision capabilities. In their work, the entire host operating system is "lifted up" and run inside a malicious VMM. This work is most closely related to ours, but there are a number of key differences. Our purpose is not simply to install a rootkit, nor do we wish to virtualize the execution of the victim OS. Instead, we are interested in both virtualizing and supervising the execution of individual applications on the victim host to help automate the discovery of faults and vulnerabilities.

The members of a DAC stealthily intercept the execution of applications on the victim host and instead run the application in a supervised environment. We have proposed binary-level application behavior profiling [19] in the context of self-healing systems; DAC members profile an application's behavior looking for errors and faults with exactly the opposite goal. The behavior of an automorphic worm is somewhat similar to a Midgard worm [16, 27].

Finally, in a parallel proposal, Raiciu *et al.* [26] propose the notion of *exploit hijacking*, where an attacker listens for alerts such as those produced by Vigilante, and automatically creates their own worm based on the exploit described in these "smart" defenses.

# 3. AUTOMORPHIC WORM DESIGN

The theme of this paper is to examine the implications of using binary supervision mechanisms from the self-healing arena for the automation of malware creation. Attackers already use similar tools to assist in identifying potential vulnerabilities. Debuggers and code-coverage tools are used manually for a small set of applications at a time. Inputs that trigger errors are manually created and correlated.

The main intent of an automorphic worm would be to create and maintain a database of discovered bugs and vulnerabilities in the applications it would supervise. It will use this portfolio to spread in a variety of ways. Of course, these worms should include a component that notifies the DAC owner about a newly discovered bug and the input that triggers it. Finally, this type of worm may also contain a polymorphic engine to help further disguise it, but this capability is not strictly necessary or definitive for an automorphic worm.

An automorphic worm has two main purposes:

1. decrease the likelihood of detection via network traffic analysis (not necessarily host-based analysis) by changing the application or vulnerability that is attacked with each generation. This type of worm is not initially written to be metamorphic, but rather becomes so with experience.

2. increase the attacker's resources by testing many different applications and configurations for errors that may lead to vulnerabilities in an automated fashion

Automorphic worms will generally contain a supervision environment that will be used to monitor a set of applications on the victim host. The Memcheck tool for Valgrind [22] is a good example; the tool can assist in detecting memory corruption errors. Each worm instance would not necessarily need to carry a full binary supervision environment when spreading. It may be able to take advantage of one already installed on the machine, or download one from the web (such an action seems relatively harmless: an *http* download or CVS checkout may be fairly common) and patch it if necessary.

The supervision environment can be selectively invoked for a subset of application runs, or it can operate each time the application is loaded. Different strategies may be appropriate for I/O bound *vs.* CPU bound applications, depending on how much of a slowdown the user would experience. Furthermore, the advantage of an Application Community idea can be well applied: the large size of the DAC can be used to collaboratively "cover" all portions of an application such that any single instance is only slowed down during a small fraction of its execution.

The goal of using the supervision environment is to discover how the application's execution is affected in response to particular inputs. These inputs can be generated by the worm instance itself, delivered from other worm instances in the DAC, supplied by the botnet owner, or simply occur during the "normal" operation of the application on that particular host.

One interesting idea is to develop the ability to do machine learning on sequences of basic blocks to see if a discovered bug or vulnerability occurs in other places in the supervised application's execution. The worm can also see if such a vulnerability exists across applications and hosts. If the automorphic worm has already identified a high-quality portfolio of vulnerabilities, finding basic blocks that match the features of "known" (to the worm and attacker) vulnerabilities is a powerful advantage. However, identifying such a feature set is a research-level problem that we are currently investigating in other work.

Finally, an automorphic worm could be made more powerful by the use of various planning techniques, although this is not a focus of our work. As an example, once a DAC node has identified an input that causes an error for a particular application, it may try several transformation strategies on that input and re-run the application on it. The point of such a feedback loop with the application as an oracle would be to generalize the particular input data or events that cause an error. Such knowledge is potentially useful in generating polymorphic variants. This type of AI technique may be of interest to an attacker, but would probably require some application or domain-specific knowledge to be of practical use.

# 4. TESTBEDS & SHIFT-BLIND ATTACKS

After infecting a machine, bot malware usually tries to disable the security applications running on the host. The list of commonly targeted security applications includes Windows XP built-in firewall and its anti-spyware technology, commercial anti-spyware tools, anti-virus applications, and security or management tools that may be used to detect, block, disable, or remove malware from the system [10].

Instead of removing the security mechanisms in a subverted host, an attacker may take a different approach: disable their ability to report malware detection both to the user and to the network and use them to monitor and analyze their reaction to newly developed exploits. Disabling the reporting ability of a security application is relatively simple: it requires that the malware is installed in a lower layer in the botted system [12]. Modern rootkits can easily monitor, intercept, and modify the state and actions of other software on the system while remaining relatively invisible.

Harnessing the power of a DAC is just the next evolutionary step: instead of testing the exploits on a single botted machine, attackers farm out the testing process to multiple zombies. This has a multitude of benefits: through its inherent diversity and heterogeneity a DAC can report detection and reaction results from a large collection of anti-malware and protection mechanisms. In addition, triggering a defense system in a botted host provides valuable information about its internal operation. Such operation includes, but is not restricted to, signature generation, false positive/negative assessment algorithms, and communications with centralized servers to obtain program and signature updates. An automorphic worm can take advantage of this information to evade and attack the defense systems.

Indeed, when a new vulnerability is discovered, update messages are transmitted to all hosts subscribed to the security service enabling them to identify the new threat. Of course, if the host is botted, the same messages become an early warning system for the attacker allowing him to "shift" to another dormant and undiscovered exploit. Since these signature and vulnerability announcements are security application specific, the attacker gains knowledge about the detection capabilities of all the security applications deployed in the botnet. Even if the botnet does not have a specific security application deployed, the attacker can potentially install the application on some of the botted hosts to obtain the output of the updates and the early warning system. A trivial example of this type of notification is the Windows automatic update "Patch Tuesday" cycle.

Finally, the automatic feedback systems present in most, if not all, of the anti-virus and anti-spyware systems can be employed to "blind" these defense mechanisms by "shifting" their attention from the real threat. We can imagine the following scenario: a DAC triggers the detection mechanisms of its security application by feeding them with either a synthetic or real exploit. As a reaction to this exploit, the security mechanisms contact their respective centralized servers to transmit information about the newly identified exploit. If the DAC is large enough (botnets of size more than 100,000 nodes have emerged [7]), and the transmission is coordinated, the central servers in charge of gathering the signatures will suffer an application-level (and perhaps also a network-level) DDoS attack. This attack will impede or completely cripple the servers' operation and render the early warning systems incapable of responding to other threats. In the meantime, the attacker will deploy the real worm so that it is concealed in the noise of the newly generated alerts. This type of attack could successfully compromise large numbers of additional nodes, since the spread of a worm or scans for new victims will be completely unobserved despite the deployed security mechanisms and intrusion detection systems.

# 5. DISCUSSION

The primary difference between a computer security professional and a cracker is mainly one of ethics – tools remain largely the same. An illustrative example of this incongruity is the `nmap` tool, but the analogy extends to many other security tools and techniques. In addition, attackers have long known that compromising a large collection of machines provides an amplification in the amount of force they can bring to bear in DDoS attacks. We have recently suggested using similar collections of machines for defensive operations. However, one of the main goals of this paper was to show how binary-level supervision (previously used for detection and self-healing) can be used to help identify vulnerabilities on behalf of an attacker. While we are not aware of any botnets that employ the techniques we describe in this paper, constructing them can be done with tools that are broadly available today. Certain aspects of an Application Community (and by extension, a DAC) are similar to those in a GRID computing environment; specifically, the command and control infrastructure for dispatching jobs must address the same challenges.

## 5.1 Distributed, Automatic Bug Reports

The intelligence-gathering mechanisms in a Dark Application Community share similarities with automated bug reporting facilities like the TalkBack plugin for Mozilla Firefox or Microsoft's crash reporting utility. Many other software systems, both open source and proprietary, often include some form of bug reporting with varying degrees of automation. While it would be difficult to obtain this type of data from Microsoft, it would certainly be worthwhile to gather statistics from open source projects on the rate of bug discovery and how quickly the corresponding error is fixed and made available for download. Such statistics would give a good starting point for evaluating the rates at which a DAC can identify potential vulnerabilities. Even if such data is subject to sources of error (*e.g.*, self-selected group reporting, refusal to report due to privacy concerns, errors that are not reported because they may be timing-dependent and rarely occur, etc.), it is a valuable resource to help assess the threat posed or reward offered (depending on one's point of view) by a DAC.

## 5.2 Challenges and Limitations

While a proof of concept DAC is straightforward to implement, there are some issues that may prove difficult in making this idea practicable. Aside from defense measures like artificial diversity (*e.g.*, address space, instruction set, or configuration randomization), the key question is whether this form of distributed bug identification produces enough leads and does not cost too much in terms of resource usage to be noticeable to the user. Furthermore, exploitable vulnerabilities may be very rare compared to all discovered bugs, so an attacker has a small space to choose from, even assuming that an exploit is easy to prepare. As one participant summed it up, the most critical issue is the question of *yield* – how many DAC machines are required to discover a given number of bugs? Moreover, it is clear that relying solely on user input to drive the bug discovery process is not enough, because users may not behave sufficiently differently to raise the yield. A similar form of this problem reduced the utility of n-version programming for introducing artificial diversity. Finally, the behavior of a DAC member is probably detectable via host-based mechanisms.

Behavior-based approaches to host-based anomaly detection are a large part of the answer for these types of attacks, but only if they are employed in an environment that can reliably observe the events that form the basis of the behavior. This requirement is equivalent to the observation made in [12] that the fundamental advantage goes to the "lower-level" system. The key problem is to classify "emulation" or "supervision" behavior and determine if this type of activity is happening to applications.

The creation of systems that achieve this capability is a serious challenge. The core capabilities essentially amount to creating signatures specific to a vulnerability. While there has been some recent work on the theory of such vulnerability signatures [3], the definition of a vulnerability is not clear, and some work [6] questions whether identifying vulnerabilities rather than control primitives is actually worthwhile.

We define a vulnerability to include two things: a set of code paths in an application that collectively represent the actions enabling an attacker to gain control via some input, and the set of configuration data, process state (*e.g.* thread timing), and application data that is outside the control of the attacker that activate these code paths.

A vulnerability-specific signature is a point on a line somewhere between exploit-specific signatures and full "behavior-based" recognition. Both signature and behavior model recognition have a spectrum ranging from regular expressions through pushdown automaton/symbolic execution to Turing Completeness. However, it is a fundamental result of computability theory that high-level system behavior is undecidable – Turing Machines are unable to *computationally* recognize (*i.e.,* decide) complex behavior. As an alternative, researchers build models that approximate the behavior and accept certain false positive and negative rates. Even in the case where behavior is simple enough to be computationally recognizable, the computation may take too long, so uncertainty is accepted there as well.

## 5.3 Future Work

Further study is needed to assess how much work such a system would save malware writers. Several experiments are possible. First, a simulation can be constructed that splits the spreading behavior of one worm into three or four different ones so that it remains under the threshold of traffic rate based detectors. Second, we can construct an Application Community based on either a VMM or a change to the OS application loader that intercepts process creation and runs each process inside a memory debugger like the Memcheck tool for Valgrind. This Application Community can gather statistics on the number of memory-based errors present in a wide variety of applications under many different inputs. Such inputs can be saved and forwarded along with the error alerts for further analysis. One important consideration for a DAC is to make sure that it is able to avoid a DoS based on submitting duplicate or bogus bug or error reports. Duplicates can easily be discarded by keeping a keyed hash database of previous reports based on important fields. Duplicate notifications are actually somewhat encouraging (if not produced maliciously) because they indicate that an error may be widespread and therefore be quite valuable if it is remotely exploitable. Bogus reports are somewhat more difficult to deal with. An approach similar to Vigilante's Self-Certifying Alerts (SCA's) is probably appropriate: an error report must contain enough information to duplicate the error or vulnerability automatically. We note that most users will not have the expertise to inject fake reports and that deliberately generating crash reports at a manual timescale effectively means continuously crashing an application, which most users won't have the patience for.

Valgrind is a popular platform for creating such security analysis tools; researchers have implemented instruction set randomization [2], function profiling [19], and taint-tracking [25] tools. Our own experience with Valgrind tools suggests that most applications remain usable, but are noticeably slower during startup and somewhat slower during operation. Slicing the monitored application up in an Application Community style should reduce the performance overhead, as each member of the DAC only does a fraction of the total work. Since the application is slowed down for a small (and potentially variable) part of its operation, the user may be able to rationalize the reduced performance by incorrectly blaming their CPU, network hardware, ISP, primary memory, operating system, the computer in general, or even other spyware or malware.

# 6. CONCLUSIONS

We have described a new class of malware based on the idea of "Collaborative Insecurity." A collection of such malware can be leveraged to perform distributed profiling of a wide set of applications in order to magnify the ability of the attacker to identify vulnerabilities and construct exploits. In an Application Community, such monitoring can assist defensive operations. Instead, a Dark Application Community employs this type of monitoring to subvert system security. The key idea is to make a collection of bots a renewable resource. For example, bots that have been identified as spam relays or DDoS zombies can be re-commissioned by the attacker as "research" bots that seek to fuzz or stress test applications for new vulnerabilities. Such behavior minimizes the overhead of botnet maintenance by finding uses for retired nodes.

Finding vulnerabilities and constructing attacks is a difficult and time-consuming process. Attackers can benefit as much as security professionals from automating the identification of vulnerabilities. Such automated identification is made easier by the very security tools that attempt to mitigate the presence of a vulnerability. Attackers can take advantage of binary-level supervision techniques to quantify the behavior of applications running on the victim hosts in the DAC. If these applications include intrusion defense systems, the DAC can be used as a testing platform to vet new exploits. The main purpose of a DAC is to support the assembly of a portfolio of vulnerabilities that the DAC malware can use to spread. The attacker can employ a *shift-blind* attack using this portfolio by shifting to a new vulnerability after blinding early warning systems with an expendable exploit. These capabilities motivate a discussion on uses for a trusted computing base and binary-level behavior tools in order to counter this type of attack.

# Acknowledgments

The authors would like to thank the initial round of reviewers for pointing out ways to deepen the discussion about automorphic worms and attacks on "smart" defenses. In

particular, we were able to refine our definition of automatic vulnerability discovery. There is some amount of disagreement as to what actually constitutes a vulnerability, and the reviewers motivated us to clarify our definition. We were also able to clarify the basic mechanism an attacker might use to automatically identify vulnerabilities. Lastly, one reviewer pointed out that a DAC is essentially one way to define the threat that botnets pose beyond sending unsolicited bulk email.

The discussion session at the workshop itself was quite productive and raised a number of issues that we have attempted to address in the final revision. One recurring line of questioning focused on the feasibility of leveraging "normal" user input and actions to trigger exploitable vulnerabilities. While we explore this question and other issues in Section 5, it is important to keep in mind that user input is but one method of driving the "testing" of applications on behalf of the attacker. We enjoyed the fruitful discussion of all aspects of this idea. Finally, thanks are again due to Carrie Gates and Bob Blakley for their detailed note-taking.

## 7. REFERENCES


[1] Anagnostakis, K., Greenwald, M. B., Ioannidis, S., Keromytis, A. D., and Li., D. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the $11^{th}$ IEEE International Conference on Networks (ICON)* (October 2003), pp. 403–408.

[2] Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanovic, D., and Zovi, D. D. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the $10^{th}$ ACM Conference on Computer and Communications Security (CCS)* (October 2003).

[3] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[4] Chinchani, R., and Berg, E. V. D. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proceedings of the $8^{th}$ International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 284–304.

[5] Costa, M., Crowcroft, J., Castro, M., and Rowstron, A. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (October 2005).

[6] Crandall, J. R., Su, Z., Wu, S. F., and Chong, F. T. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the $12^{th}$ ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[7] Dagon, D. The Botnet Trackers. *Washington Post* (February 2006). `http://www.washingtonpost.com/wp-dyn/content/article/2006/02/16/AR20060%21601388.html`.

[8] Du, W., and Atallah, M. J. Secure Multi-Party Computation Problems and their Applications. In *Proceedings of the New Security Paradigms Workshop* (September 2001), pp. 11–20.

[9] Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In $10^{th}$ *ISOC Symposium on Network and Distributed Systems Security (SNDSS)* (February 2003).

[10] Ianelli, N., and Hackworth, A. Botnets as a Vehicle for Online Crime. `http://www.cert.org/archive/pdf/Botnets.pdf`, December 2005.

[11] Kim, H.-A., and Karp, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference* (August 2004).

[12] King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J., and Lorch, J. R. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[13] Krugel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the $8^{th}$ International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 207–226.

[14] Krugel, C., Toth, T., and Kerer, C. Decentralized Event Correlation for Intrusion Detection. In *Proceedings of the International Conference on Information Security and Cryptology (ICISC)* (December 2001).

[15] Leyden, J. DDoSers attack DoubleClick, *http://www.theregister.co.uk/20 04/07/28/ddosers_attack_doubleclick/* , July 2004.

[16] Li, J., Ehrenkranz, T., Kuenning, G., and Reiher, P. Simulation and Analysis on the Resiliency and Efficiency of Malnets. In *Proceedings of the $19^{th}$ Workshop on Principles of Advanced and Distributed Simulation (PADS 2005)* (2005).

[17] Lincoln, P., Porras, P. A., and Shmatikov, V. Privacy-Preserving Sharing and Correlation of Security Alerts. In *Proceedings of the USENIX Security Symposium* (2004), pp. 239–254.

[18] Locasto, M. E., Sidiroglou, S., and Keromytis, A. D. Application Communities: Using Monoculture for Dependability. In *Proceedings of the $1^{st}$ Workshop on Hot Topics in System Dependability (HotDep-05)* (June 2005).

[19] Locasto, M. E., Stavrou, A., Cretu, G. F., Stolfo, S. J., and Keromytis, A. D. Quantifying Application Behavior Space for Detection and Self-Healing. Tech. Rep. CUCS-017-06, Columbia University, 2006.

[20] Malan, D. J., and Smith, M. D. Host-Based Detection of Worms through Peer-to-Peer Cooperation. In *Proceedings of the $3^{rd}$ ACM Workshop on Rapid Malcode (WORM)* (November 2005).

[21] Moore, D., Shannon, C., Voelker, G., and Savage, S. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference* (April 2003).

[22] Nethercote, N., and Seward, J. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science* (2003), vol. 89.

[23] Newsome, J., Brumley, D., and Song, D. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* $13^{th}$ *Symposium on Network and Distributed System Security (NDSS)* (February 2006).

[24] Newsome, J., Karp, B., and Song, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2005).

[25] Newsome, J., and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The* $12^{th}$ *Annual Network and Distributed System Security Symposium* (February 2005).

[26] Raiciu, C., Handley, M., and Rosenblum, D. Exploit Hijacking: Side Effects of Smart Defenses. In *Proceedings of the SigCOMM Workshop on Large Scale Attack Defense (LSAD)* (September 2006).

[27] Reiher, P., Li, J., and Kuenning, G. Midgard Worms: Sudden Nasty Surprises from a Large Resilient Zombie Army. Tech. Rep. CSD-TR040019, University of Oregon, 2006.

[28] Richardson, T. Cloud Nine blown away, blames hack attack. `http://www.theregister.co.uk/content/archive/23770.html`, January 2002.

[29] Robertson, W., Vigna, G., Kruegel, C., and Kemmerer, R. A. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceedings of the* $13^{th}$ *Symposium on Network and Distributed System Security (NDSS)* (February 2006).

[30] Singh, S., Estan, C., Varghese, G., and Savage, S. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[31] The Cambridge-MIT Institute. DoS-Resistant Internet Working Group Meetings, February 2005.

[32] The Honeynet Project & Research Alliance. Know your Enemy: Tracking Botnets. `http://www.honeynet.org`, March 2005.

[33] Toth, T., and Kruegel, C. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)* (2002).

[34] Wang, C. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, 2000.

[35] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM* (August 2004).

[36] Wang, K., and Stolfo, S. J. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* $7^{th}$ *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2004), pp. 203–222.

[37] Yegneswaran, V., Barford, P., and Jha, S. Global Intrusion Detection in the DOMINO Overlay System. In *ISOC Symposium on Network and Distributed Systems Security* (February 2004).

[38] Young, A., and Yung, M. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley and Sons, February 2004.