MALPITY: Automatic Identification and Exploitation of Tarpit Vulnerabilities in Malware

Sebastian Walla, Christian Rossow
CISPA Helmholtz Center for Information Security
s8sewall@stud.uni-saarland.de, rossow@cispa.saarland

***Abstract*—Law enforcement agencies regularly take down botnets as the ultimate defense against global malware operations. By arresting malware authors, and simultaneously infiltrating or shutting down a botnet's network infrastructures (such as C2 servers), defenders stop global threats and mitigate pending infections. In this paper, we propose malware *tarpits*, an orthogonal defense that does not require seizing botnet infrastructures, and at the same time can also be used to slow down malware spreading and infiltrate its monetization techniques. A tarpit is a network service that causes a client to stay busy with a network operation. Our work aims to automatically identify network operations used by malware that will *block* the malware either forever or for a significant amount of time. We describe how to non-intrusively exploit such tarpit vulnerabilities in malware to slow down or, ideally, even stop malware.**

Using dynamic malware analysis, we monitor how malware interacts with the POSIX and Winsock socket APIs. From this, we infer network operations that would have blocked when provided certain network inputs. We augment this vulnerability search with an automated generation of tarpits that exploit the identified vulnerabilities. We apply our prototype MALPITY on six popular malware families and discover 12 previously-unknown tarpit vulnerabilities, revealing that all families are susceptible to our defense. We demonstrate how to, e.g., halt Pushdo's DGA-based C2 communication, hinder SalityP2P peers from receiving commands or updates, and stop Bashlite's spreading engine.

I. INTRODUCTION

Recent malware outbreaks such as the WannaCry ransomware [48], Mirai [3] or Bashlite [2] (both Linux-based botnets known for their DDoS capabilities of over 620 gigabits per second [21]) demonstrate that we lack mechanisms to cease the spreading of epidemic malware incidents. On the one hand, several methods exist to defend against malware, such as host-based [5], [31], [33], [34] or network-based [23]–[25], [50] malware detection. On the other hand, many of these defenses are local to the scope where they are applied, and thus only protect individual hosts or networks. In contrast, defenses that cripple entire botnets at the infrastructural level (such as botnet takedowns [46], [49], [52], [53]) are inherently more complex, if not infeasible (e.g., for reputation-based peer-to-peer botnets such as SalityP2P [49]). Worse, such takedowns require the cooperation of providers that operate malicious network infrastructures. Yet until such global defenses are effectively applied, malware will continue its nefarious activities.

In this paper, we propose *malware tarpits* as a novel defense paradigm to stop malware operations. During its entire lifecycle, malware heavily relies on network communication. As demonstrated by WannaCry, Mirai and Bashlite (and several additional incidents in the older past), worms scan the Internet for victims to spread rapidly by exploiting vulnerabilities remotely. Furthermore, malware receives instructions from command-and-control (C2) servers over the network. In addition, to monetize itself, malware typically launches network-based attacks such as email spamming, harvesting, or Distributed Denial-of-Service (DDoS) attacks. Our idea is to identify whether we can use a tarpit to slow down or pause such communication, and by doing so, stop or at least slow down the malicious interactions of a botnet. A tarpit is a network service that feeds or withholds network inputs to a client (in our context, the malware) such that the client gets (temporarily or indefinitely) stuck in its network operation. For example, if a client calls the `recv()` socket operation without having checked that data can be read, it will be put into waiting state until data in the receive buffer is actually available to be read. A tarpit that does not feed any inputs will thus block the client process indefinitely. Tarpits thus allow us to decelerate or even stop malicious network behavior in malware. In the malware context, tarpits were originally introduced by Tom Liston to slow down the spread of the CodeRed worm [35]. He created the *LaBrea* tarpit that replied with TCP SYN/ACK segments to scanning hosts. Liston leveraged the fact that the scanner's timeout for the half-completed connection was sufficiently long to keep the worm busy waiting for some time. When entering the tarpit, the worm thus has fewer resources to scan other (real) hosts, which slightly slows down spreading. This general concept of network tarpits is completely non-intrusive, as it neither causes the client to crash, nor creates busy loops, nor requires executing attacker-injected code.

Unfortunately, several challenges were left open by Liston's initial proposal. First, in order to develop his tarpit, Liston had to manually analyze and understand the network behavior of the worm. Given the steady rise of new malware variations, this is infeasible to do in a timely way in practice. Especially for worms, tarpits have to be deployed *instantly* to be effective, as otherwise the population will have grown to a critical mass. Second, Liston's proof-of-concept idea (and also other tarpit implementations) only *slows down* a client's interactions. Instead, we aim to also search for code patterns that can be exploited with what we will call a *sticky tarpit*, i.e., one that *blocks* the malware for its entire lifetime. Third, tarpit vulnerabilities are not only specific to the spreading engine of malware. In fact, the more complex malware becomes, the more likely it is that code patterns in other network-dependent modules (e.g., a spam engine, or C2 communication) can be blocked. Finally, Liston's idea and follow-up works to integrate such tarpits into the Linux firewall [60] primarily aim to abuse tarpit-like behavior in the kernel's TCP/IP stacks (e.g., TCP retransmissions, small TCP window sizes, etc.) and are thus

limited to TCP, whereas tarpit vulnerabilities also generalize to UDP communication.

We propose a generic methodology to identify tarpit vulnerabilities in malware that tackles the aforementioned limitations. Our methodology (i) is completely automated, (ii) covers both sticky and slowing tarpits, (iii) is not constrained to tarpits in a malware's spreading engine, and (iv), identifies UDP/TCP socket-level tarpits based on the malware's interaction with the socket API that go beyond existing tarpits that focus on particularities of the TCP/IP stack. Our approach does not require access to the malware source code. Instead, by observing the dynamic behavior of a malware program, we can reason about whether or not the malware communication is susceptible to tarpit attacks. To this end, we monitor both the POSIX socket API and Windows' Winsock API and search for blocking network operations. Assuming we can control inputs to these operations, we then extract code patterns that allow us to tarpit the malware. Finally, we fully automatically generate ready-to-use code that exploits the identified vulnerabilities.

We prototype this methodology in a tool called MALPITY. MALPITY takes a malware sample as input, executes it, reports on tarpit vulnerabilities, and automatically generates code to exploit the tarpit. We apply MALPITY on six recent and widespread malware families (Bashlite, Pushdo, Pynamer, SalityP2P, Tofsee, and WannaCry) and identify 12 previously-unknown tarpit vulnerabilities. Seeing that all families are susceptible to tarpits suggests that the potential of our defense is far higher than one would intuitively assume. Law enforcement agencies and/or network telescope and honeypot operators can use these automated tarpits to mitigate malware. Tarpits thereby help to contain the spreading of worms before their population grows to a critical mass, to seize control over existing infections, or to stop malicious activities of malware (such as spamming, fraudulent ad clicks, or SEO fraud).

Our contributions can be summarized as follows:

- We generalize the concept of tarpits in malware. We show that tarpits can not only be abused to slow down scanning, but in fact, can also be generalized to sticky tarpits and affect more than just the spreading engine.
- We introduce a generic methodology to monitor the socket API(s). From this monitoring, given an arbitrary program as input, we then infer and search for code patterns that result in tarpit vulnerabilities.
- We implement a fully-automated prototype MALPITY, which we will release to trusted parties upon request.[1] We demonstrate that tarpit vulnerabilities are present in all six prevalent families we investigated.

II. METHODOLOGY

Malware inevitably relies on communication, e.g., to perform malicious activities (such as sending spam emails), to exchange data with C2 servers, or to spread by infecting other parties. Our overall goal is to slow down or even stop these malicious activities by creating *malware tarpits*. Such tarpits bring malware into a state in which it is ideally infinitely, but at least temporarily, stuck in a socket operation. Malware that has entered a tarpit can thus not (or only slowly) continue its harmful operations. We will briefly introduce a running example, and based on this, explain our general methodology.

A. Running Example

Consider our running example in Figure 1, which demonstrates the `readUntil` function of Bashlite, which is vulnerable to a tarpit. In line 11, `select` checks whether a blocking socket named `fd` has a non-empty read buffer. If so, Bashlite continues and reads a one-byte command from `fd` using `recv` (line 14). In line 19, Bashlite checks if the initial byte is 255 to identify Telnet communication. If so, in line 20, Bashlite aims to receive more data to complete the command. Note, however, the tarpit vulnerability in line 20. As Bashlite calls the second *recv* without checking if yet another byte is available in the read buffer, it may face situations where no such data is available. The previous call to *select*, if successful, has only checked whether *at least one* byte was available to read. As a result, the caller thread would get stuck forever in the second receive operation unless data is served. That is, given that the socket is blocking, the second *recv* call would bring the thread into a state where it blocks indefinitely. This, which we will introduce as a *sticky* tarpit vulnerability, makes Bashlite's spreading engine vulnerable to abuse. As soon as a scanned target—randomly chosen by Bashlite—deploys a tarpit that abuses the vulnerability, it would stall Bashlite's scanning engine and thus stop its spreading. Anyone who happens to be probed by a Bashlite-infected system can thus interrupt one worm instance. The chances of being probed in the early stage of a worm are significant for large networks (e.g., in a `/8` IP telescopes).

B. Attack Surface for Tarpits in Malware

We will now generalize from this running example. First, to assess the impact of tarpits, we will group them according to the logical malware component in which they are located. Nowadays malware is typically divided into several threads that run independently from each other. It likely will not be possible to lure all threads into a tarpit. If only selected threads can be tarpitted, we would like to evaluate the potential impact of such a tarpit. Following the modular design of malware nowadays, we thus categorize the following three malware components in which tarpits can have a high impact:

1) Command-and-Control (C2): The C2 logic is responsible for receiving commands from a botmaster and feeding back data (e.g., stolen passwords or credit card information). At first sight, tarpits in this component are not that helpful, as defenders would have to control the C2 server to feed inputs that trigger the tarpit. However, successful botnet takedowns in the past have forced malware authors to deploy highly redundant C2 endpoints. This increases chances for defenders to get contacted by a malware. For example, malware might iterate over a list of servers, from each of which it tries to receive commands. Alternatively, malware may use Domain Name Generation algorithms (DGAs) [4], [47] to deterministically generate C2 domains for locating C2 servers. In the extreme, malware replaces infrastructures by fully-decentralized peer-to-peer (P2P) networks for C2 communication [49].

[1] To request access to MALPITY's source code, please send both authors an email that clearly states your institution and your goal of using MALPITY. Students should include their supervisor in the email.

```
1  int readUntil(int fd, char *toFind, int matchLePrompt, int timeout, int timeoutusec, char *
       ↪ buffer, int bufSize, int initialIndex){
2
3          int bufferUsed = initialIndex, got = 0, found = 0;
4          fd_set myset;
5          unsigned char *initialRead = NULL;
6
7          while(bufferUsed + 2< bufSize && (tv.tv_sec > 0|| tv.tv_usec > 0))
8          {
9                 FD_ZERO(&myset);
10                FD_SET(fd, &myset);
11                if (select(fd+1, &myset, NULL, NULL, &timeoutstruct) < 1) break;
12
13                initialRead = buffer + bufferUsed;
14                got = recv(fd, initialRead, 1, 0);
15
16                if(got == -1 || got == 0) return 0;
17
18                bufferUsed += got;
19                if(*initialRead == 255){
20                      got = recv(fd, initialRead + 1, 2, 0);
21                      /* ... */
22                }
23         }
24         /* ... */
25 }
```

Fig. 1. Code of the `readUntil` function in Bashlite. There is a tarpit vulnerability due to an unguarded repeated `recv` call in line 20.

Either way, the malware contacts a number of C2 endpoints that are potentially controlled by defenders. For example, when DGAs are used, defenders can mimic a C2 server by registering a domain name that will be generated in the near future (e.g., see Torpig [54]). Controlling an endpoint is even easier in P2P botnets, which are open in nature and allow defenders to add fake bots. To defend against such spurious C2 endpoints, malware typically deploys authenticity checks before processing commands. However, tarpits can force vulnerable malware to keep communicating with spurious servers *before* commands are checked, and thus help to prevent malware reaching genuine C2 endpoints.

2) Attack Engine: Besides the C2 component, malware may also include attack engines that are used for monetization. For example, the attack engine may send spam (spambots), crawl websites (harvesters), or execute Denial-of-Service attacks (DDoS bots). If malware is vulnerable to a tarpit in its attack engine, the attacked endpoints can slow down or even stop the malware's attack. The chance of getting contacted on the first try by the malware to abuse the tarpit vulnerability is fairly small. But consider a spambot that aggressively iterates over a range of millions of mail exchanges to send spam. If a single of those email relays were to abuse the tarpit, this would significantly slow down (or even stop) the malware's spam engine. Similarly, if bots crawl the Web, defenders just need to control any of the sites that the malware contacts.

3) Spread Engine: Finally, tarpits in the spread engine can help to slow down (or even stop) malware distribution. Worms especially take an aggressive approach and aim to spread to other non-infected hosts in order to increase their population. Most prominently, malware scans the Internet for vulnerable or otherwise easy-to-abuse services. For example, Mirai and Bashlite search for SSH/Telnet-enabled devices, and then bruteforce a small set of hardcoded credentials. If we can find tarpits in the malware's spread engine, we can mitigate worm epidemics, regardless of the exact spreading mechanism.

Summary: We identified three malware components that are susceptible to tarpits. Given the sheer complexity and number of malware samples nowadays, manually finding tarpits in these components requires a substantial amount of reverse engineering and expertise. This motivated us to find an automated mechanism to identify tarpits in a given malware binary. Note that we did not aim for an automated method to classify in which malware component a given tarpit is situated. Once the challenging task of identifying a tarpit vulnerability in the malware is solved, assigning such vulnerabilities to their logical component does not require much manual effort—especially if we can collect metadata (such as content, endpoint information, or ports) about the susceptible connections.

C. Malware Tarpits

After introducing where tarpit vulnerabilities can be located, we will now classify the tarpit according to its type. To this end, we generalize our introductory example and introduce two kinds of malware tarpits: *sticky* and *slowing* tarpits. Sticky tarpits are the strongest form and bring a program (or more precisely, the current thread) into a blocking state in which it stays *indefinitely*. This corresponds to the example in Figure 1, in which the malware would get stuck in the receive operation forever. In contrast, a slowing tarpit vulnerability only temporarily blocks a program for some finite time. For example, if socket operations use a certain timeout, the stuck execution will continue as soon as the timeout expires.

Our goal is to automatically identify such tarpits in malware programs. We thus carefully iterated over the socket

TABLE I. TARPIT-YIELDING PATTERNS

Action	Condition	Protocol
Receiving	kernel receive buffer empty or contains too little data	UDP/TCP
Sending	kernel send buffer full	TCP
Peeking	no socket in ready state	TCP/UDP
Connecting	no SYN/ACK received	TCP

API, which is the underlying basis for any kind of user-space communication in both Windows and Linux, to search for socket interactions that introduce tarpits. Table I summarizes the four tarpit patterns we identified. This list represents a (likely) non-exhaustive but powerful set of coding patterns that enable tarpits. We will describe them in the following.

1) Receiving Data: When a program attempts to receive data over a blocking TCP/UDP socket for which no data is available, the receiving thread blocks. Technically, both `recv` (TCP) and `recvfrom` (TCP/UDP) return only if the kernel buffer for received data is not empty or if the connection is closed. In addition, receiving data is even blocking if the buffer is non-empty, but the caller asks to wait until a certain number of bytes have arrived (e.g., via the `MSG_WAITALL` flag). To trigger either tarpit vulnerability, the remote side simply stops sending data, and blocks the receiving side forever. The code in Figure 1 includes such a tarpit. A receive timeout will only occur if explicitly configured (e.g., using `ioctl`, or the Windows counterpart `setsockopt`), turning this otherwise sticky tarpit into a slowing tarpit.

2) Sending Data: In contrast to receiving tarpits, tarpits in send operations are specific to TCP, as UDP send operations usually do not block[2]. If the program tries to send something over a blocking TCP socket, but the kernel buffer for sending data is full, the send call blocks until space in the buffer is available [43]. In Windows 10, the default send buffer is 8 kB [44], which the remote party can fill by not acknowledging received data. As soon as TCP segments are acknowledged, their data is deleted from the send buffer, therefore making space for new data to send. If the remote party does *not* acknowledge the data, the kernel will give up sending the data. After a few retransmission attempts, the connection will be aborted and any blocking send call will return. Sending tarpits are thus usually only slowing, bounded by the maximum time the kernel may store unacknowledged data.

To compute the slowdown, we have to inspect the retransmission algorithm a bit more closely; this slightly differs per operating system. Windows 10 uses five TCP retransmissions by default. The initial retransmission timeout is determined by a smoothed round-trip time. On a typical connection, this timeout is about 50ms. TCP's exponential backoff algorithm then multiplies this initial timeout per trial i by 2^i. The total timeout for a typical send attempt is therefore $\sum_{i=0}^{5} 50ms * 2^i = 3150ms = 3.15s$. While this seems low, it may be possible to stack several send calls to amplify the slowdown.

3) Peek Functions: To avoid send/receive operations blocking the caller, the POSIX socket API introduced functions that make it possible to peek at whether data can be read from or written to blocking UDP/TCP sockets. `select` is the most popular example. Peek functions can be called with a user-defined timeout, which has a large impact on potential tarpit vulnerabilities. The peek function would block until the timeout (if any) is hit, or one of the sockets becomes ready for sending/reading—whatever occurs first. If we can control the state of all sockets probed by the peek functions, we can make sure a socket never enters the "ready" state. This way, we can either abuse a long timeout (slowing tarpit) or even leverage the fact that no timeout was specified (sticky tarpit).

4) Connection Establishment: In TCP, establishing a connection can already introduce a slowing tarpit vulnerability if the caller did not specify a timeout before. For example, `connect` blocks the caller until the connection is successfully established or the connection attempt times out. This timeout is determined by the number of allowed retransmissions of SYNs. Windows uses two SYN retransmissions by default, but again doubles the timeout (by default initially 3 seconds) after each retransmission [42]. The entire timeout for a connection attempt hence defaults to $\sum_{i=0}^{2} 3 * 2^i = 21$ seconds.

D. Vulnerable Socket Operation Patterns

We will now use our knowledge of blocking network operations to infer which of a malware's socket interactions are susceptible to a tarpit. To this end, we assume control over *if* and *what* a remote communication partner (the tarpit) sends to the malware, to *any* socket. This over-approximation may assume control over sockets that are not controllable by defenders in practice. For example, if the malware uses a single hard-coded C2 server, exploiting a tarpit in the C2 communication would require controlling the server. Our methodology will thus also reveal tarpit vulnerabilities that cannot be (easily) exploited. We do not see a *reliable* generic method to exclude such uninteresting tarpits. While one could generally distinguish C2 from non-C2 communication [28], Section II-B outlined usable tarpits even in C2 communication. We argue that manual analysis aided with automated socket metadata (e.g., traffic content, domain name, port number) quickly reveals if an endpoint can be controlled by a defender. As we will show, the interesting tarpits outweigh the uncontrollable ones, and are not only located in C2 communication, but also in spreading modules and attack engines.

Using this threat model, we will search for execution paths that lead to tarpit vulnerabilities triggered by blocking sockets. To this end, we have to monitor the socket state by tracking operations performed on a socket, as each call may change the blocking semantics of the socket. In the following, we thus define generic code patterns that represent execution paths that are vulnerable to tarpits. Formally, we require that (i) the socket is in blocking mode, and (ii) that the vulnerable execution path satisfies at least one of the following actions:

- A connect operation on a TCP socket.
- A receive operation that is not directly preceded by a peek function that ensures data to be available on this socket. Note that any receive operation voids the guarantee of data availability for subsequent receive operations on the same socket (see running example).

[2]Technically, the remote party has no control over the fill status of the send buffer, and thus cannot force this buffer to fill up.

- A receive call that is issued with a flag to wait until a certain amount of data is received, but the required amount is not assured to be readable.
- A peek function is called either with a "high" timeout argument (slowing), or without a timeout (sticky).
- One or more calls to sending socket operations, which in their sum try to send more than the kernel's send buffer can hold. Note that this is regardless of whether peek functions were used, as they do not guarantee non-blocking sending behavior for multibyte inputs.

E. Identifying Vulnerable Execution Paths

After having defined all vulnerable code patterns, we will now discuss methodologies for identifying them in a given (malware) program. To decide if a socket operation can fall into a tarpit, we must know the underlying sockets and their state. We will briefly discuss the different options we can leverage.

Static Analysis: The first option that comes to mind in order to search for such code paths is static program analysis. However, unless we can use perfect data flow analysis and assume non-obfuscated code, tracking sockets statically across functions or even processes quickly becomes infeasible. First of all, static analysis would fail for malware that obfuscates its code, e.g., using packing [6], [41]. But even if we assumed non-obfuscated malware, standard commercial static analysis tools such as IDA would not be able to track control flow (and thus socket operations) across jump tables or more complex function pointers (such as C++ vtables). Finally, even if such problems were solved, static analysis may identify paths that have unsatisfiable path constraints, or those that are never triggered during runtime.

Selective Symbolic Execution: To solve the problem of unsatisfiable paths, we could leverage symbolic execution. Symbolic execution [30] considers all inputs to be variable, giving us the possibility to learn which inputs would trigger tarpit vulnerabilities. However, using naïve symbolic execution, we would face path explosions, increasing the required execution time. Furthermore, due to the immense amount of possible functions we would have to emulate (e.g., Windows's/Linux's software stack and libraries), it is impractical to model all of those functions' effects using symbolic variables. Instead, we first experimented with Selective Symbolic Execution (S2E) [15], which allows us to define the sources for symbolic data and will execute all other instructions that depend only on non-symbolic data concretely. This narrows down the number of symbolically-executed instructions to a minimum and speeds up the analysis. In early experiments, we treated all data which is received via sockets as a symbolic source. This reflected our assumption that all data from remote communication partners is controllable. However, this already led to path explosions, e.g., due to non-numeric inputs that have to be parsed.

Concolic Execution: As a way out of the path explosion problem, we experimentally marked all data received over the network as concolic. Concolic is a portmanteau of concrete and symbolic, meaning that upon a constraint-driven fork, the search engine should choose the branch which the concrete value satisfies first, but can also choose another path that can be symbolically satisfied. Initial results were promising and concolic execution would be our preferred choice for malware that uses plaintext communication. Yet concolic execution still cannot traverse code paths with network inputs that are decrypted/deobfuscated before further processing. The decryption/deobfuscation adds a lot of constrains on the concolic input, which need to be solved by the SMT solver, when checking the feasibility of a path. However, this requires a lot of time and causes the analysis to get stuck in the decryption/deobfuscation routines.

Dynamic Analysis: Seeing the problems caused by other alternatives, we chose concrete malware execution as the way to go forward. In dynamic analysis, we execute the malware and wait until it executes paths that include interesting socket operations. This methodology works even for complex input parsers and any kind of manipulation function (decryption, deobfuscation, decoding), and even survives runtime unpacking.

On the downside, dynamic analysis only receives a subset of all possible program inputs, and thus misses those code parts that never become active. In our concrete use case, however, the negative impact of missing code is minimal. In fact, to successfully tarpit malware, we would have to target code parts that regularly become active. Put differently, those code parts that our dynamic code analysis misses will likely also not become active for other malware-infected hosts, and are thus less (if at all) useful for tarpit operations.

To find tarpit vulnerabilities using dynamic analysis, we observe all socket operations performed by the malware. Whenever a socket is created, we start to monitor its state and observe all operations on it. This way, we can track whether or not a socket is blocking whenever a potentially blocking network operation is performed. Similarly, we can monitor whether peek functions guarantee that a certain socket has data/space in its according buffer. If we encounter a situation in which a blocking operation is performed *without* previously guaranteeing that the operation does not block, we have identified a tarpit vulnerability.

F. Automated Exploit Generation

As a final step of our methodology, we automatically generate a program that exploits a tarpit vulnerability that we have identified. For example, when monitoring receiving functions, we save the received data corresponding to each socket. When we find a socket to be vulnerable, we create server-side code that automatically sends the previously recorded data up to the point the tarpit vulnerability was discovered. The exploit sends the data in the exact order in which packets were received earlier, but modifies/omits the first message that would trigger the tarpit vulnerability. The tarpit is generic, meaning that it allows any client to connect (or to send messages in the case of UDP). Once a malware contacts the generated tarpit, the tarpit thus replays the inputs such that it will enter the same execution path as under dynamic analysis. This effectively traps the malware in our tarpit.

Recall our running example from Figure 1 (page 3). In order to reach the tarpit in line 20, Bashlite first has to receive the single byte `0xFF`. If nothing else is sent, Bashlite is stuck indefinitely. The corresponding exploit, which our tool

```
1 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
3 s.bind(('0.0.0.0', 6666)) # DNAT redirects any port to this one
4 s.listen(123)
5 con, client_address = s.accept() # accept client
6 con.setblocking(0) # non-blocking socket to avoid tarpitting the tarpit
7 con.sendall(b'\xff') # send payload to reach tarpit vulnerability
```

Fig. 2. Automatically generated tarpit code to exploit Bashlite's `readUntil` tarpit vulnerability (Figure 1, page 3) for a single client.

generated, is shown in Figure 2. Upon a connection request it accepts the connection, sends the required payload and keeps the connection open without sending any other data. The server socket port is arbitrary, as we use DNAT to redirect all traffic to an arbitrarily chosen port (in our example, TCP/6666).

III. Implementation

We prototyped the described methodology in a framework called MALPITY that searches for tarpit vulnerabilities in Windows-based malware samples. In the following, we will describe implementation details that go beyond the methodological description of the previous section.

A. Dynamic Analysis

We first had to choose an environment that allows us to dynamically analyze malware. To this end, we selected S2E [15], which uses QEMU as an underlying type-2 hypervisor. Although its original purpose is symbolic execution, S2E is a convenient choice as it allows us to hook functions in a transparent way. QEMU translates basic blocks of the guest's executable code to basic blocks of the host's executable code during runtime. Chipounov et al. modified this to translate x86 code to LLVM's intermediate representation, when at least one of the instructions in the basic block references symbolic data. The LLVM code is then symbolically executed using KLEE [13], unless basic blocks do not depend on symbolic data. As our initial experiments demonstrated the problems of symbolic executions, we switched to using S2E's concrete execution model by marking no data as symbolic. Lacking any symbolic data, S2E executes the entire program concretely. We still greatly benefit from S2E's ready-made function hooking engine that we can leverage to monitor the socket APIs.

B. Function Hooking

To track potentially vulnerable execution paths and the state of sockets, we hook calls to network operations. We use S2E code translation for this, which can be instrumented to generate events to which plugins can subscribe. In order to hook instructions, S2E instruments all instructions of a tracked module to trigger an event upon their execution. On execution, it checks whether the instruction pointer matches one of the instructions which is supposed to be hooked. If so, S2E calls the callback function that was registered as a hook.

S2E monitors all call and return instructions to hook functions. We thus just have to specify the addresses of call targets to monitor. S2E accepts relative virtual addresses that survive relocation. That is, we can simply encode fixed addresses of the dynamically-linked functions that we target. Yet call-based hooking does not work for jump tables, although they are frequently used by programs (`switch` statements, vtables in C++ programs, etc.). Since jumps are not monitored, S2E would not detect such function calls. To solve this, we use S2E's instruction hooking to hook the very first instruction of each function. Similarly, we inspect the function's return value by registering a return hook.

We furthermore retrieve the arguments passed to the hooked socket API functions. We do this in our hooks that are placed before the first instruction of each monitored function. The 32-bit Winsock API uses the *stdcall* calling convention, in which arguments are pushed from the right to the left [36]. Thus, before the function's prologue, the stack pointer points to the return address (the top most element). We retrieve function argument arg_i from address `esp+4*(i+1)`, i.e., from the stack just before the return address. While the 64-bit calling convention is different, all the malware in our experiments (like malware in general) still uses 32-bit Winsock.

Hooking is implemented at the hypervisor level, and is thus transparent to the guest. This also means that the malware cannot detect nor disable hooks. Furthermore, the hooks are executed on the host and manipulate the guest's memory via virtual machine introspection (VMI). Virtualizing the malware execution guarantees strong process isolation between the malware and our analysis. We thus chose this approach over a less stealthy implementation that places function calls in user space. Having said this, similar to any other (dynamic) malware analysis environment, malware can in principle still detect the fact that it runs in a virtualized or sandboxed enviroment [20], [62]. Providing a fully stealthy analysis environment, such as those proposed in Ether [17], is out of scope of this work.

Our dynamic hooks also allow us to deal with packed malware that aims to thwart static analysis [56], even if malware relies on virtual machines to obfuscate code [1]. Regardless of the packing mechanisms, our dynamic analysis environment will notice if a malware loads a library to execute network operations, and installs hooks accordingly. In fact, four of six malware families in our evaluation will be packed.

C. Overview of Hooked Socket API(s) Functions

We will now detail which functions we hook in order to find tarpit vulnerabilities in malware. Most importantly, we have to verify whether a socket is blocking and if the socket can send/read data without being blocked. To this end, we monitor all relevant library calls, which we identified by manually investigating the list of exported functions in the

TABLE II. OVERVIEW OF HOOKED FUNCTIONS AND THEIR INFLUENCE ON THE ANALYSIS

Category	Name	Effect
Socket Creation/ Termination	closesocket shutdown socket WSASocket	Creates a socket, connects and terminates it, respectively
Alter Blocking	setsockopt WSAAsyncSelect WSAEventSelect WSARecvDisconnect	Changes blocking mode of socket
Peeking, Alter Blocking	ioctlsocket WSAIoctl	Can either change blocking mode of socket or guarantee its readability
Receiving	ReadFile recv recvfrom WSARecv WSARecvEx WSARecvFrom	Removes readability guarantee of socket
Peeking	select WSAPoll	Can guarantee readability of socket
Sharing	WSADuplicateSocket	Shares socket with another process

Winsock library (`ws2_32.dll`). Table II lists the 19 functions we classified as relevant, all of which MALPITY hooks on Windows systems. Targeting Windows, we have to capture both the POSIX socket API as well as the Windows Socket API (Winsock; all API functions prefixed with `WSA`). Most of the Winsock equivalents of POSIX socket functions (e.g., `socket` vs. `WSASocket`) behave semantically equivalently, but offer a few extra options. Whenever necessary, we captured the subtle differences between the two APIs.

Note that higher-level networking APIs are implicitly captured by monitoring these lower-level socket functions. For example, consider `ReadInternetFile`, which is used to retrieve resources from the Internet. Internally, it calls lower-level functions such as `WSARecv`. This is similarly true for other higher-level functions such as `InternetOpenUrl`, `FtpGetFile` or `HttpSendRequestA`. It is therefore sufficient to hook the lower-level functions to also capture the behavior of higher-level APIs.

In the following, we detail the motivation and specifics behind tracking exactly these functions.

1) Socket Creation and Termination: First and foremost, we have to track which sockets are created. To this end, we hook the `socket` and `WSASocket` function, both of which create and return a socket. After creation, a TCP client socket would then connect to a remote target. Even for UDP sockets, although optional, calling `connect` or similar functions would bind the socket towards one particular target. Regardless of the socket type, however, we do not need to track the connection targets. Instead, our automatically generated exploit code will run on an arbitrary target and service port, and we will use destination Network Address Translation (DNAT) to redirect all other ports. Tracking connections might become important if tarpits span multiple sockets, and then suddenly the order between network operations might become important—an aspect we left open for future work.

When sockets and their connections are terminated, we track this and clean up internal state to eliminate potential false positives. If `closesocket` is called, we remove all information associated with the corresponding socket identifier. `shutdown` can be used to close the connection in sending/receiving or both directions. Similarly, `WSARecvDisconnect` closes the connection in the receiving direction. Any subsequent attempt to receive data on such a socket will not block and simply returns no data. Similarly, trying to send data on sockets closed for sending will also not block.

2) Blocking Mode: By default, a socket is in blocking mode, i.e., all write/receive operations will block until they succeed. In our model, upon socket creation, we thus assume a socket is blocking. The blocking status can then be changed by using `ioctlsocket`, `WSAIoctl`, `WSAEventSelect`, or `WSAAsyncSelect` [45]. If `ioctlsocket` is given first the socket, next the `FIONBIO` command and then some value as the last argument, it changes the blocking mode of a socket. If the last argument is non-zero, the socket is set to non-blocking; otherwise the socket is set to blocking. `WSAIoctl` works analogously, except that the blocking flag argument is wrapped in the `inBuffer` pointer. Finally, both `WSAEventSelect` and `WSAAsyncSelect` implicitly set a socket to non-blocking mode, which we also track.

A slight variation of non-blocking sockets is blocking sockets with a preconfigured timeout. For example, it is possible to specify a timeout for blocking receive calls after which they should return with an error if still no data arrived within the predefined timeout. The single function to control this timeout is `setsockopt`, which we therefore also hook to record its respective timeout. Whenever such timeouts are specified, tarpit vulnerabilities cannot be sticky, as the malware will (eventually) return from the blocking operation.

3) Functions for Receiving Data: To find tarpit vulnerabilities for read operations, we search for paths in which the malware tries to receive data from a blocking socket without ensuring that data actually can be read. When any receiving socket API function returns, we record the size and the content of the received data for a possible exploit generation.

To determine if the receiving call blocks, we need to know if the socket is guaranteed to hold enough readable data. This is the case if the socket was checked in a peek function to be readable beforehand. The readability guarantee of the peek function no longer holds if a subsequent receiving function was already called on the socket. However, with every rule there is a exception: If the receiving function was called with a flag which does not remove the data from the kernel buffer (e.g. `MSG_PEEK`), the readability guarantee of the peek function still holds.

Receiving functions that should wait until the buffer is completely filled (e.g., using the flag `MSG_WAITALL`) can still block if not enough data is available. For those we therefore check how much data the peeking function guarantees to be readable. For `select` and `WSAPoll` one byte; for

ioctlsocket, together with the FIONREAD command, it is potentially more than one byte.

Technically, we hook the functions recv, recvfrom, WSARecv, WSARecvFrom, and WSARecvEx to capture all receiving socket API functions. WSARecv and WSARecvFrom have the same blocking semantics as recv, unless the socket is overlapped and asynchronous completion routines or structures are specified. Such a completion routine would be executed once the data has been received.

In addition to the explicit receive operations, we similarly hook ReadFile if the handle in the arguments is a socket handle. Again, as mentioned initially, we do not need to track those higher-level API functions like InternetOpenUrl or FtpGetFile that use the basic socket primitives internally.

4) Peek Functions: Peek functions make it possible to check if data is writable/receivable for a socket. Consequently, they can determine if a receive operation creates a tarpit vulnerability. For example, consider the select function, which takes as input three sets of file descriptors (sockets), for which it checks whether at least one byte can be read from or written to them, respectively, or if sockets are in some exception state. Upon return, the respective sets only contain those descriptors which can immediately be read or written to, or which triggered an exception. Thus if a program uses peek functions, e.g., to check if data can be read before actually calling the receive call, it carefully avoids tarpit vulnerabilities. We thus track peek functions, in particular those that check whether sockets can be read. Once select returns, we iterate through its readable set and check which sockets are guaranteed to have non-empty read buffers.

For WSAPoll, each socket is augmented with flags which determine if the socket is to be checked for readability or writability. After return, similar flags state the readability, and so forth. On return we iterate over the sockets passed to WSAPoll and mark those sockets guaranteed to have data, which have the readable flag set.

As it could happen that malware executes a peek function, but ignores whether the function guarantees a socket to be readable, we also want to simulate the peek function not guaranteeing any socket to be readable. Therefore, upon the execution of a peek function, we fork the current state. In one state we just monitor the readable sockets as described above. In the other state, we simulate that no socket is readable. For select, we achieve this by emptying the read state file descriptor set. We also empty the exception state file descriptor set, as otherwise malware would simply close those file descriptors, because of their apparent error state. For WSAPoll we achieve the same by zeroing the sockets' readability output flags.

In contrast to select and WSAPoll, ioctlsocket can guarantee more than one byte to be receivable. When ioctlsocket is given the FIONREAD command, it is used to check for the number of bytes in the receive buffer. If ioctlsocket guarantees at least one byte to be readable, we mark the socket as guaranteed to have data. WSAIoctl works analogously to ioctlsocket, except that the number of readable bytes is wrapped in the outBuffer pointer.

While the receiving part of peek functions can be easily monitored, the writing side is harder. Technically, whether or not a socket is writable depends on the space in the socket's send buffer maintained by the kernel. Similar to receiving data, a socket is writable if at least one byte can be sent. However, in contrast to receiving functions, the write function may block if the provided data exceeds the space in the socket's send buffer. The same is true for WSAPoll, which marks the socket to be writable under the same conditions. Unfortunately, the state of the send socket's buffer depends heavily on the kernel, which maintains the state of the TCP connection. In the best case, sending data will block only until the kernel determines that the previously sent data cannot be delivered. Although the remote party can control what data it acknowledges, eventually, if not all data to be sent is acknowledged, the kernel will abort the socket and resume all blocking operations. We therefore consider neither write operations, nor their related peek functions. That is, we assume that write functions (or peek functions checking for writable sockets) do not block. We leave a thorough evaluation of the full potential of sending tarpits open for future work and will discuss them in Section V.

In addition to their effects on subsequent receive/send operations, peek functions themselves are blocking and can thus also result in tarpit vulnerabilities. As we consider all remote communication partners to be controllable by us, checking for a tarpit vulnerability created by a select or WSAPoll call is straightforward. For both functions, we have to ensure that sockets are only checked for receivability. For select, we check whether the timeout pointer and the write and exception set are null, resulting in a sticky tarpit. Similarly, for WSAPoll, negative timeout values result in sticky tarpits. If the timeout value is "high" (at a configurable threshold) for either peek function, this results in a slowing tarpit.

5) Functions for Sending Data: To capture sending tarpits, one would have to hook all sending API functions (send, sendto, WSASend, WSASendTo, WSASendMsg, WSASendDisconnect and WriteFile). We could then track the number of already sent bytes per socket by accumulating the number of bytes sent per sending API function call. As soon as the socket's send buffer in the kernel is full, sending would then block temporarily. However, given the limited utility of sending tarpits (cf. Section II-C), we did not extend MALPITY with such functionality. We will discuss an extension of our work in that respect in Section V.

6) Shared Sockets: Processes can share a socket. For example, if one process calls a peek function to guarantee that the socket has available data, it can notify another process to receive the data. Technically, it is challenging to track sockets shared between processes. Up to now, we could identify a socket solely by its identifier, but now one also needs to know in which process a function is called and whether the socket is shared. Unfortunately, socket identifiers are not guaranteed to be the same across processes. Therefore, per socket in one process, we maintain the set of tuples (pid, socket) that represent the socket identifiers in other processes referring to the same socket. Whenever one changes an attribute of a socket, we now also iterate over this set and update the attribute for the respective socket identifiers in the other processes.

On Windows, sockets can be shared by calling WSADuplicateSocket on sockets that should be shared.

The function requires specifying the target process which should receive the shared socket and returns a pointer to a `ProtocolInfo` structure. A pointer to this structure (or a copy of it) is then passed as an argument to `WSASocket` to get an identifier for the shared socket in the target process. Each `ProtocolInfo` structure has a unique identifier, which we will call `uid`. Technically, we thus map (`uid`,`targedPid`) → (`srcPid`,`socket`).

Similarly, in the other process, we have to determine the source of the shared socket. We thus retrieve the `srcPid` and `socket` identifier using our mapping created in `WSADuplicateSocket`. We obtain the `uid` from the arguments to `WSASocket` and the `targetPid` argument of the mapping is the current Pid. Using those two, we build and maintain the tuple (`srcPid`,`socket`) for each shared socket.

D. Differences from the Socket API of Linux

In general, our approach is not platform dependent, and can also be applied to other operating systems such as Linux. That is particularly true for environments that use the previously described POSIX socket API. Technically, we would need to update the hooking targets, as the socket API function addresses differ per operating system. In addition, there are some subtle differences in the semantics of the socket API. For example, Linux adds a flag called `MSG_DONTWAIT` which allows specifying that a specific socket operation should not block (rather than configuring the entire socket as non-blocking). The Linux `fcntl` function, which can be used to modify or retrieve socket options, translates to `ioctlsocket` and `WSAIoctl`. While we did not port our prototype to Linux, doing so would be mainly an engineering effort, leaving a working setup of S2E (which currently supports both Windows and Linux) as the main challenge.

E. Malware Containment

When dynamically analyzing the malware, we give the bot Internet access such that it can freely expose its behavior. This setup allows as a live C2 server to instruct the malware with commands, which triggers the malware's malicious behavior. Having said this, a malware with unconstrained Internet access can harm others, e.g., due to its spreading or attack capabilities. Following best practices [51], we thus limit how the malware can communicate with the outside world. First, we can deploy strict rate limiting to throttle the up- and download bandwidth, e.g., to mitigate Denial-of-Service attacks or scans initiated by the malware. This way, we rigorously limit the number of targets a malware can contact, and the amount of traffic it can send to each—without losing the generality of our methodology. Second, we can redirect critical services to internal hosts, such as outgoing SMTP connections and probes for exploitable systems listening on standard services (e.g., NetBIOS). Redirecting these services—instead of dropping the traffic—allows us to simulate the expected behavior, and thus trigger larger fractions of the malware's networking code. Third, we collaborate with the local network administrators and network service provider to learn about potential breaches. This setup has been shown to work well in the past. Only the fact that malware sometimes also connects to sinkholes during the dynamic analysis has created alerts about potential abuses in our network.

IV. Evaluation

We have implemented MALPITY as an operable prototype that allows us to examine the utility and accuracy of our new method of identifying tarpit vulnerabilities. This section will describe the malware we used for our evaluation, the evaluation setup, and the detailed evaluation results.

A. Dataset

To measure the effectiveness of our proposed methodology, we applied MALPITY on the following set of six prominent malware families. The malware samples were selected from a public malware sandbox and were verified to become active at the time of analysis. Appendix A lists the SHA256 hashes of the precise malware samples that we analyzed, all of which can be obtained via VirusTotal or upon request.

- **Bashlite**: The Bashlite worm (a.k.a. *LizardStresser* or *Gafgyt*) appeared first in 2014 and abused the Shellshock vulnerability to infect over a million BusyBox devices. Bashlite thus represents a whole class of other Linux worms that quickly followed, such as Mirai or Hajime, which reportedly infected several million Internet-of-Things devices [3]. Bashlite spreads by abusing weak or default credentials of Telnet or SSH-enabled devices.

  To fit our Windows-based evaluation setup, we had to port Bashlite to Windows. Since there is no complete forking equivalent in Windows, we removed `fork()` operations in Bashlite, resulting in a single-threaded program. Note, however, that MALPITY is not limited to single threads, but can also monitor multiple threads. We will discuss the original subprocess of the Bashlite tarpits we found in our qualitative assessment. Furthermore, as the Bashlite source code does not contain active C2 servers, we implemented a C2 server that sends random commands to its clients. To mitigate the harm of DoS attacks, we used localhost as the target of attack commands. We also replaced Bashlite's generation of a random public IP in the scanner with the localhost address. We set up a Telnet service listening on port 23, which after accepting would constantly send the byte `0xFF` (following standard Telnet behavior).

- **SalityP2P (v4)**: SalityP2P is a peer-to-peer based botnet that has operated since 2007 and is known to withstand takedown attempts due to its decentralized scheme [49]. SalityP2P acts as a dropper and uses both UDP and TCP communication to interact with its peers. The overall SalityP2P population was estimated to be over a million peers in 2013 [49]. Since then, many SalityP2P peers migrated from the older version (v3) to the newest (v4), and still form an active botnet. SalityP2P spreads as a file infector, yet does not expose any network-based spreading activity.

- **Pynamer**: Pynamer is a malware downloader that has been undergoing steady development for years. In its recent incarnation, Pynamer uses a JSON-like protocol to obtain tasks, and is mainly used to drop spambots on the infected host. Pynamer does not spread.

- **Pushdo**: Pushdo became popular as a malware downloader and for years has been one of the most nefarious and largest botnets around. Pushdo is famous for its tight alliance with the spambot Cutwail, which is responsible for huge portions of global spam overall. Like Pynamer, Pushdo is also not a worm.
- **WannaCry**: WannaCry is a recent malware outbreak attacking vulnerable SMB/NetBios implementations in Windows. In May 2017, it infected over 200,000 systems worldwide with ransomware. WannaCry acts as a worm and scans the Internet for further systems that could be exploited similarly as the current host.
- **Tofsee**: Tofsee is a popular botnet used for spamming. It downloads spam templates and recipient lists from its C2 server and then connects to the mail servers to send spam emails via SMTP. Unless specific modules are dropped, Tofsee does not attempt to spread to other systems by default.

B. Evaluation Setup

We created a 64-bit Windows 10 Enterprise version 1703 virtual machine using QEMU, in which the malware is executed. The virtual machine has 2 GB RAM and a simulated Intel Core 2 Duo CPU with 2.40GHz. We executed each piece of malware separately for a maximum of 60 minutes. We reset the virtual machine after each execution, such that all malware samples start in the same clean environment.

We restricted our evaluation to those network APIs that can, in principle, result in sticky tarpit vulnerabilities. We thus ignored sending and connecting network operations, as discussed in Section III-C. This way, we reduce the number of execution paths that we have to verify manually as part of our evaluation, and obtain a small number of high-impact tarpits we can qualitatively assess. Most identified tarpit vulnerabilities are thus sticky and guaranteed to keep a thread busy forever. Sometimes, in particular if timeouts are used, we can nevertheless encounter slowing tarpits, which we then also considered.

We gave special treatment to peek functions. Given that their main purpose is to wait until any socket is "ready", they naturally incur a desired timeout. We thus omitted all peek functions that yield only a slowing tarpit, as such behavior is pretty much standard. This still leaves those peek functions that are called without a timeout.

C. Results

We applied MALPITY to all six malware families mentioned above. In order to evaluate its effectiveness, we invested a significant amount of effort to reverse engineer the malware samples in order to manually identify all tarpits in each executable. We used manual backward slicing, starting at socket API invocations, to find a ground truth of tarpit vulnerabilities manually. In the following, we will compare our manual findings with the results we obtained via MALPITY.

Table III sums up our overall results. We added a dash ("–") if a component did not contain a tarpit vulnerability, and add "n/a" if the entire component did not exist. MALPITY automatically found and generated exploits for twelve tarpit vulnerabilities in all analyzed malware samples, five of which even contain sticky tarpits. Most importantly, MALPITY found *all* tarpit vulnerabilities contained in our ground truth. Recall that we dropped the idea of symbolic execution and thus only monitor those paths that are "visited" during dynamic analysis. The result thus indicates that tarpit vulnerabilities are typically located in functionality that is quickly explored during concrete execution with a live C2 server.

TABLE III. TARPIT VULNERABILITIES REVEALED BY MALPITY.
"N/A": COMPONENT DOES NOT EXIST IN THE MALWARE.
"–": NO TARPIT VULNERABILITY EXISTS IN THE COMPONENT.

Malware	C2 Comm.		Attack Engine		Spreading Engine	
	sticky	slowing	sticky	slowing	sticky	slowing
Bashlite	1/1	–	–	–	1/1	–
Pushdo	1/1	1/1	–	1/1	n/a	n/a
SalityP2P	2/2	–	–	–	n/a	n/a
Tofsee	–	1/1	–	1/1	n/a	n/a
WannaCry	–	–	–	–	1/1[3]	–
Pynamer	1/1	–	1/1	–	n/a	n/a

The vulnerabilities are furthermore located in critical malware functionality. In particular, the identified tarpits make it possible to (i) block Bashlite's spreading engine, (ii) stall SalityP2P's update mechanism, and (iii) capture Pushdo bots that contact DGA-generated C2 servers. Furthermore, we generated slowing tarpits in WannaCry, Pushdo and Tofsee, all of which can slow down their spreading or attack campaigns. In the following, we will describe the tarpits in more detail.

1) Bashlite: MALPITY found the two sticky tarpit vulnerabilities, one of which was in Bashlite's C2 component, and the other in its spreading engine. Effectively, the spreading engine represents our running example in Figure 1 (page 3). To trigger this tarpit, Bashlite first connects to one of its C2 servers. When it receives a command to scan, it forks a new child process which then connects to at most 512 random IP addresses via Telnet (TCP port 23). After being connected successfully to one of these potential infection targets, Bashlite will first check if data is receivable, and if so, receive one byte. If this byte is equal to `0xFF`, meaning the follow-up byte is a Telnet command, Bashlite attempts to receive another two bytes, but this time without peeking for data beforehand. Since the socket is blocking, the scanning target can thus trigger a sticky tarpit vulnerability and hinder the scanner from reaching other hosts for all eternity. Interestingly, there can be at most one scanning child process per Bashlite instance [2], so Bashlite would stop spreading once stuck in this tarpit. While stalling the current population of Bashlite, this tarpit can only assist in a takedown, as the rest of Bashlite stays functional.

The second tarpit is in the C2 component. When trying to receive commands from its C2 server, Bashlite peeks only once to see if data is available (using `select`), but then calls `recv` in a loop until the buffer is filled. The previous `select` call only guarantees that at least one byte of data is available, which is consumed after the first `recv` call. Subsequent `recv` calls are not guaranteed to have data available and may therefore block. Thus, as soon as defenders control one of the Bashlite

[3]Technically this is a sticky tarpit, however the parent thread terminates the thread containing the sticky tarpit after 1 hour.

C2 servers, they can capture Bashlite-infected systems.

2) SalityP2P: Given its P2P nature, SalityP2P's C2 component interacts with other (potentially defender-controlled) peers in the network. SalityP2P contains two sticky tarpit vulnerabilities in this communication, one of which can be abused. The abusable vulnerability is located behind a decryption function. When running SalityP2P in concolic mode, even with no forks allowed, we were not able to get through the decryption routine. When running the malware concretely, MALPITY found both vulnerabilities in less than 7 minutes.

The abusable tarpit is in SalityP2P's update mechanism. SalityP2P starts multiple threads, which each have their own purpose. One thread T_{upd} is responsible for checking that the saved peers are still up and if they have an update. This thread spawns, every forty minutes, up to seven concurrent threads T_i. Each of these T_i then checks if the neighboring peer is online and whether it has an update. If the remote peer has an update, T_i will try to receive the update on a blocking socket without checking that data is receivable beforehand. This results in a sticky tarpit vulnerability and therefore enables blocking T_i indefinitely. T_{upd} will wait until all its spawned threads have returned, and will thus also be affected by the tarpit. As this P2P mechanism is the only way to retrieve updates for the vast majority of 85% of SalityP2P bots hidden behind a NAT gateway [49], the tarpit can be abused to prevent SalityP2P peers from receiving updates. In fact, receiving "URL packs" (lists of URLs with malicious modules to download) is the only type of command, except for peer propagation, exchanged by SalityP2P [19]. Thus, the bot master would lose control over bots as soon as they are trapped in the tarpit. Therefore, this tarpit can be used to takedown 85% of the SalityP2P botnet. Note that the assumption of controlling at least one peer per bot is reasonable and has been demonstrated to work in other contexts, such as using sensors in P2P networks [49].

The other (non-exploitable) sticky tarpit that MALPITY found is in the component that is responsible for handling incoming version checks from other peers. Here, a `recvfrom` call deliberately blocks a thread until any of the remote peers sends a UDP datagram to the UDP server socket. However, the UDP server socket is available to all peers, and—in contrast to TCP, unless specifically specified as such—is not bound to a particular connection. Defenders thus have no control over who will send data to this socket. As soon as either party sends a UDP segment, the thread will continue.

3) Pynamer: MALPITY found both sticky tarpit vulnerabilities in Pynamer. Pynamer will request new tasks every thirty seconds. To this end, it contacts a hard-coded C2 server and queries a list of tasks via HTTP. The server will eventually respond with a list of URLs the malware should contact next. Upon reading this response from the server, the first tarpit vulnerability appears. However, the generated tarpit is not as useful, as it requires control over a hard-coded C2 server.

The other tarpit is more interesting and is part of Pynamer's attack engine (in the case of downloaders, the attack engine simply fetches and installs further malicious files). After receiving URLs from the server, the malware contacts each URL. Here, Pynamer exposes its second tarpit vulnerability, as again data is received on blocking sockets without peeking for data first. This may give defenders a bit more control than just a C2 tarpit, given that malware authors consider file hosting infrastructures less critical than C2 servers. Since Pynamer is single-threaded, one could use this second tarpit to take down the Pynamer population.

4) Pushdo: Pushdo contains three tarpits, all of which MALPITY found. The first, slowing, tarpit is located in a thread which tries to receive the SMTP greeting from hard-coded mail servers as an online check. The tarpit has a 30-second timeout per mail server, but is located in a "while true" loop, which runs forever in this thread. As only this thread is stuck, this tarpit can only reduce the rate of connection attempts.

Pushdo also contains a slowing tarpit with a 5-second timeout in a `recvfrom` call, while performing its custom name resolution with a DNS resolver. The receiving operation is called in an infinite loop that only exits if less or more than 600 bytes were received. Assuming control over the DNS resolver that Pushdo uses, one could abuse this slowing tarpit. A tarpit could always answer with exactly 600 bytes, forcing Pushdo's according thread to never leave this loop, and preventing it from further DNS resolution.

Finally, Pushdo contains a sticky tarpit on a global socket that is used for C2 communication. To keep in contact with the C2 server despite takedown attempts, Pushdo has several fallback mechanisms for its C2 communication [8]. One fallback mechanism is its DGA [16] that generates C2 domains. One can force Pushdo to use its DGA-based C2 communication by taking down its hard-coded C2 IPs and domains. Whereas registering *all* DGA-generated domains quickly does not scale, we can now abuse the tarpit instead. Defenders can now register a single domain name that is or will be generated by the DGA. Once the domain points to the tarpit, it traps bots forever, effectively taking the botnet down.

5) WannaCry: MALPITY found a vulnerability in WannaCry's spreading engine. The spreading engine uses up to 128 threads (T_{probe}), each of which generates and then contacts random IP addresses. For each reachable IP address, T_{probe} probes all addresses within the /24 subnetwork (excluding 0 and 255 as the last octet) with a timeout of one second per probe. If a host is reachable, T_{probe} starts a new thread $T_{connect}$, which then connects to the IP on TCP port 445 (SMB). WannaCry interacts with the peer and repeatedly tries to receive data from the remote communication partner. The worm uses `recv` without peeking for data availability. As the socket is blocking, and has no timeout assigned, this blocks $T_{connect}$.

Interestingly, WannaCry employs a watchdog functionality. The parent thread T_{probe} will terminate the child thread after one hour, and then continue connecting to subsequent addresses in the same network. This illustrates how an originally sticky tarpit can turn into a slowing tarpit in practice. After $T_{connect}$ terminates (or is shut down), T_{probe} will try the next IP address of the /24 subnetwork. Thus, T_{probe} will again start $T_{connect}$ to connect to the next IP address, and again wait, for the thread to terminate for at most one hour. Effectively, this means that defenders cannot abuse this tarpit to stop an attack, but instead, to slow down the spreading epidemics. We will further investigate the utility of such slowing tarpits in a WannaCry case study in Section IV-D.

6) Tofsee: Tofsee contains two slowing tarpit vulnerabilities, both of which MALPITY found.

The first vulnerability occurs when Tofsee receives data from the spam target's mail server. The spam engine calls `recv` in a for loop until 500 bytes have been received (or the connection is closed), semantically similar to a `recv` with the flag `MSG_WAITALL`. However, none of those receive calls on blocking sockets is preceded by peek functions. Interestingly, Tofsee can (optionally) configure a receive timeout for sockets. The timeout is determined by a configuration received from the C2 server, typically set at 30 seconds [29]. A mail server tarpit could thus launch a slowing tarpit. In fact, the tarpit can reply byte-by-byte and abuse the fact that each `recv()` call will start the timeout anew—resulting in an overall slowdown of 250 minutes per SMTP connection. The tarpit is thus suitable to slow down the otherwise aggressive nature of spambots, and can reduce the overall spam volume emitted by the botnet. Note that Tofsee receives encrypted commands from its C2 server to send mail, which again undermined our initial attempts to concolically execute the malware.

The second vulnerability enables a slowing tarpit in Tofsee's C2 communication. Again, Tofsee calls `recv` in a loop until a certain number of bytes have been received from the C2 server. Although the socket is blocking, a default timeout of 40 seconds per `recv` call is set. Again we can reply byte-by-byte to the recv calls, but this time the length the data to be received is controlled by previously received data from the C2 server. Unfortunately, as Tofsee uses a single hard-coded C2 server, this tarpit vulnerability is not useful in practice.

Note that we could not verify whether Tofsee's spreading component [29] has vulnerabilities, as the respective module was not downloaded during our evaluation.

D. Relevance of Slowing Tarpits: A WannaCry Case Study

From a defender's perspective, sticky tarpits are preferred over slowing tarpits, as they trap the malware indefinitely. However, as we have shown, it is not always possible to identify sticky tarpits. Either tarpits are just slowing from the start, or turn into slowing ones in the larger context. As demonstrated in WannaCry, even sticky tarpits can sometimes only temporarily trap threads until they are restarted. One might thus wonder whether it is worth attacking the malware and how much slowdown one might be able to achieve with slowing tarpits. This question has to be answered on a case-by-case basis, as it depends on the timeout, the nature of the thread containing the tarpit, and the number of parallel threads.

We will now illustrate the utility of slowing tarpits with the most recent malware in our dataset: WannaCry. Recall that WannaCry had a slowing tarpit in its spreading engine. We thus want to estimate the expected time until we receive at least one probing packet from a WannaCry-infected host. Each of the 128 scanning threads generates random IPs, where for the first octet they exclude numbers greater than or equal to 224 (reserved for multicast), and the number 127 (reserved for loopback communication)—leaving 223 possibilities overall. For all other octets, WannaCry skips bytes 0 and 255—presumably intended as under-approximation to exclude the network IDs and broadcast addresses.

We use the probabilistic method of Moore *et al.* [39] to compute the time for WannaCry to reach a defender's network, and then the time a defender can slow down WannaCry. To this end, we need to know the probability of our network to be chosen at random. The overall number of IP addresses that WannaCry generates at random is $254^3 \cdot 223$. We assume we own a /8 network, resulting in 254^3 IPs that WannaCry can scan. Note that such network sizes for IP telescopes are not unrealistic—several IP telescopes in use by researchers [3], [38] are of the same size. Using this /8 network, the probability to own a single IP address randomly generated by a WannaCry-infected system is thus:

$$P(contacted) = \frac{254^3}{254^3 \cdot 223} = \frac{1}{223}$$

While the chance of being contacted per random draw is low, the chances significantly increase over time. To compute chances over time, we need to know how many connection request packets (without retransmissions) per second (rate r) WannaCry sends. This can be limited by several factors such as bandwidth, congestion, or workload of the CPU. The rate furthermore depends on if WannaCry is successful in contacting an IP (and then probes the entire /24 network), or whether the IP address was non-responsive. To give an intuition, we measured WannaCry's connection request ratio for twenty minutes on a virtual machine having a single core and 4 GB RAM with 200 Mbps and 12 Mbps download and upload bandwidth, respectively. We observed 30,874 connection requests during that period, i.e., a rate r of about 26 attempts per second.

As the generation of the connection requests to a network is independent of each other, the probability to receive at least one packet in T seconds is given by a Bernoulli trial with probability:

$$\begin{aligned} P(contactedAtLeastOnce) &= \sum_{i=0}^{r \cdot T} \binom{r \cdot T}{i} \cdot p^i \cdot (1-p)^{r \cdot T - i} \\ &= ((\frac{1}{1-p})^{r \cdot T} - 1) \cdot (1-p)^{r \cdot T} \\ &= 1 - (1 - P(contacted))^{r \cdot T} \\ &= Z \end{aligned}$$

Solving this equation for the elapsed time T in seconds before at least one connection request is seen with probability Z is given by the following equation[4]:

$$T = \frac{log(1-Z)}{r \cdot log(1 - P(contacted))}$$

We can apply this computation in our concrete context. The elapsed time until a /8 network would see at least one packet from a WannaCry-infected host with 95% probability would be less than half a minute:

$$T = \frac{log(1-0.95)}{\frac{30874}{60 \cdot 20} \cdot log(1 - \frac{1}{223})} \approx 25.9s$$

[4]Note that Moore *et al.* [39] have solved $P(contacted) = 1 - (1-p)^{rT}$ slightly wrong for T in their original paper.

Once one IP address in our network is reached, WannaCry will connect to all other hosts in the same /24 network. While a probe is aborted by the watchdog after one hour, the overall probing time quickly accumulates. In fact, having 254 probed IP addresses per /24 network, we can extend the probing thread waiting time to 254 hours, i.e., ten and a half days.

For a single scanning thread, the connection rate r_t is about 0.20 hosts per second ($r/128$). Therefore the time until the thread would contact us at 95% probability is about 55 minutes. To compute the slowdown factor of the scanning thread, we divide the number of hosts the thread contacts during the time until it finds our tarpit by the number of hosts it would contact without tarpit attempts.

$$Slowdown = 1 - \frac{\frac{30874}{128 \cdot 20 \cdot 60} \cdot 60 \cdot 55 + 254}{\frac{30874}{128 \cdot 20 \cdot 60} \cdot 60 \cdot 60 \cdot (254 + \frac{55}{60})} \approx 0.995$$

At 95% probability, it is thus possible to slow down the scanning rate of a probing thread by a factor of 99.5%. This demonstrates the utility of slowing tarpits.

V. Discussion

In this section, we discuss limitations of our approach and outline potential future directions for our research.

Other Use Cases: Whereas we applied our methodology only on malicious programs, our general approach can likewise be used to identify tarpit vulnerabilities in benign software. MALPITY does not require access to source code, and hence even closed-source programs can be analyzed. If source code is provided, one can incorporate the vulnerable code patterns listed in Section II into compilers, which could then statically identify and warn about potential tarpit vulnerabilities.

Asynchronous Communication: Our current model focuses on synchronous socket communication. MALPITY therefore cannot handle asynchronous callbacks with specified callback routines, such as `WSARecv` and `WSARecvFrom`. Not only is finding tarpits in asynchronous communication more challenging due to the event-based nature, but the chances of finding exploitable tarpits in asynchronous handling functions are also lower. We thus argue that supporting synchronous socket operations will capture most abusable tarpits. Note that our approach explicitly supports threaded malware, even if the communication between the threads is not synchronized.

Send Operation Tarpits: Our prototype ignores send operations due to the strict TCP retransmission timeout handling in Windows. We have, however, not investigated what happens if the malware rapidly fills the buffer, while the tarpit acknowledges the reception of data only byte-by-byte. Similarly, sending tarpits might become interesting if they are executed in a loop. We leave both investigations open to future work.

Server-Side Tarpits: We have demonstrated the prevalence of tarpit vulnerabilities in the malware's client-side code. A blackbox-based approach, orthogonal to ours, could search for server-side tarpits that are part of the C2 server implementation. Given the complete lack of program information, the complexity of such blackbox identification is, however, significantly higher.

Statically Linked Libraries: When hooking the socket API, we assume that it is located in dynamically-linked libraries. Recently, Linux-based malware in particular started to statically link standard libraries to be compatible with multiple platforms. Our method could be adopted to such practice. Instead of hooking fixed library addresses, we would have to use code signatures to search for the functions we would like to hook [7], [26], [57].

Evasion: Malware commonly tries to evade dynamic analysis, e.g., by detecting emulation or sandboxes [37], [62]. QEMU in particular introduces artifacts such as registry keys [55]. S2E adds extra artifacts, e.g., background services, registry keys and otherwise-undefined x86 instructions. Such evasion attempts could be mitigated by using hardened and stealthy debuggers on bare-metal systems. Similarly, assuming that malware checks for such artifacts only in its packed form, one could analyze the disarmed unpacked binaries instead.

Coverage: As with every methodology based on dynamic analysis, ours also faces the fact that concrete execution will never expose the entire program functionality. As such, our analysis is not sound, i.e., it may miss tarpit vulnerabilities (false negatives). For example, if a vulnerable code part is in a subroutine that is only triggered if a certain command is received from the C2 server, our dynamic analysis would have to wait for this command—or otherwise miss the vulnerability. In our initial experiments, we used symbolic/concolic execution to resolve this problem, but with limited success, given that encoding/encryption routines quickly caused path explosions. While we acknowledge the lack of full coverage, we also argue that this is not a major problem in our use case. We aim to find tarpits in code that is commonly executed by malware, such that the impact of a tarpit is maximized. As such, tarpits in networking code that *is* executed are exactly the type of high-impact vulnerabilities we would like to find.

Exploit Serving: If malware were to have different execution paths leading to a tarpit vulnerability but each requiring a different payload, we would need to know which payload to serve, once the malware contacted the tarpit. This would be a problem if the observable behavior of the malware toward the tarpit was indistinguishable. However, none of the evaluated executables contained a tarpit vulnerability which would require a different payload depending on an indistinguishable state.

Malware Updates: Malware authors might notice the exploitation of tarpits and could remove the vulnerabilities. In fact, closing most of the tarpit vulnerabilities we found is rather straightforward, and requires either specifying timeouts, or refactoring of the networking code. Having said this, from the attacker's perspective, the consequences of tarpitting might be severe even for one-shots. For example, a sticky tarpit that deters bots from receiving updates until the malware is restarted (typically upon reboot) may decrease the malware population for a long time period. Similarly, especially for worms, automatic identification and exploitation of tarpit vulnerabilities can effectively slow down a worm's initial spreading rate. In addition, tarpitting could be combined with orthogonal botnet takedown measures. For example, sinkholing of peer-to-peer botnets works best if bots do not contact each other, as they would otherwise self-repair the defender-modified peer lists—appropriate tarpits, such as the one in SalityP2P, can thus assist

in takedowns. And finally, assuming malware code leaks on the Internet, it frequently leads to the fact that several less-skilled adversaries spawn individual botnets based on the leaked initial code base (e.g., as done for Bashlite and Mirai). Buggy source code would therefore propagate to other malware instances, such that tarpits can be reused for each botnet that was derived from the vulnerable code base.

Abuse of MALPITY: If we publish MALPITY as open source, malware authors might use it to identify tarpit vulnerabilities themselves. Similarly, MALPITY might be abused by miscreants to identify vulnerabilities in benign software. To increase reproducibility of our results and to foster future research, we thus decided to release the source code of MALPITY to trusted parties upon request, raising the bar solely for attackers to obtain MALPITY.

VI. RELATED WORK

The general idea of tarpits was introduced by Tom Liston [35], who developed the *LaBrea* software that accepts TCP connections halfway. This concept was leveraged to trick CodeRed, a popular worm that went viral in 2001, into a slowing tarpit. Since then, the idea of tarpits was not further studied for malware, potentially due to the lack of automation to identify tarpits. Also, it was widely assumed that tarpits could only slow down malware spreading and could not be generalized beyond CodeRed. We show that both identification of tarpits and exploit generation can be automated, and reveal that sticky tarpits are prevalent even in today's widespread malware. Furthermore, we expand the current state of tarpits to sticky tarpits that can make a process get stuck forever. We demonstrate the utility of socket-level tarpits that go beyond traffic shaping at the TCP/IP level [35], [60]. UDP/TCP socket-level tarpits utilize the blocking behavior of the socket API, which allows for sticky tarpits stalling an entire thread.

Since their inception, tarpits have primarily been applied in the anti-spam context. Hunter et al. [27] and Eggendorfer [18] suggest to insert server-side delays in the SMTP dialog to slow down the delivery of mail. Even if spammers violate such delays, they risk being detected as such, as benign clients (unless pipelining is used) do not follow this behavior. All these approaches focus on keeping a connection alive as long as possible, or aim to confuse scanners by randomly aborting connections. Instead, our idea targets the general blocking behavior of the socket API to slow down or even to block a program using application specific socket-level tarpits.

In the last decade, disruptive defenses against botnets have largely focused on methodologies and attempts to monitor and to "take down" botnets. Such operations typically result in *sinkholing* attempts, where defenders aim to gain control over the C2 infrastructures used by malware [10], [49]. When sinkholing is accompanied with law enforcement activities against the malware authors, it frequently becomes effective in disrupting botnets [32], [49], [58]. Our work is complementary to sinkholing, and can either assist in sinkholing, or sometimes even replace it. Sticky tarpits suddenly allow sinkholing even those networks that use highly redundant C2 infrastructures. For example, consider a DGA-based botnet, where previously it was necessary to register all time-dependent DNS domains to sinkhole the malware. With tarpits, it is sufficient to register a small number of domains, and then "clutch" the bots once they enter the blocking tarpit (such as in Pushdo). Our exploit generation assists in preparing such defenses on a timely basis and in an automated way.

Our proposed methodology is also loosely related to malware analysis methodologies in general. While we performed a case study to deploy slowing tarpits in IP telescopes (unused but routed IP addresses), others have used such networks to passively monitor the magnitude of worms [3], [38], [61]. Similarly, dynamic analysis is widely used in other contexts, such as in malware sandboxes [59] or to obtain an understanding of the malware landscape [12], [22]. Our initial attempts to symbolically execute malware come close to the idea of Moser et al., who used system snapshots and alterations of the environment to influence the control flow decisions and thus capture all possible malware behaviors [40]. Similarly, Brumley et al. mark network inputs as symbolic to find network-based triggers, like commands from a C2 server, for the malwares behavior [9]. Augmenting MALPITY with similar techniques may help to identify even those tarpits that are hidden in otherwise non-executed execution paths.

Finally, our focus on socket-based tarpits is orthogonal to attempts to exploit the complexity of an input-dependent algorithm or to trigger program crashes. Chang et al. presented an approach to detect DoS vulnerabilities in C programs that were caused by structural programming mistakes. For example, they can find faulty loop or recursion terminations that depend on network input [14], and can hang a program. Similarly, Burnim et al. use concolic execution to find inputs that trigger the worst-case complexity of a program [11]. While such vulnerabilities can also be used to tarpit a program, they are more intrusive (e.g., create a busy loop), as they target launching complex computations at the client, rather than just blocking it. Furthermore, both approaches require access to the source code, which is not always available for malware.

VII. CONCLUSION

Existing defenses against malware are either host-based (anti-virus) or take over control of a botnet's C2 communication (botnet takedowns). While host-based defenses are limited to individual hosts, botnet takedowns require complex preparations and assistance of network operators from various countries. We have shown that malware may be susceptible to tarpit attacks that allow remote parties to temporarily cease or significantly slow down the botnet operation. MALPITY can reveal socket-level tarpit vulnerabilities by monitoring the socket API and using dynamic malware analysis. Furthermore, we show how to automatically generate exploits for such tarpit vulnerabilities. We evaluated our automated analysis on six popular malware families and found tarpits in all of them. Most families were even vulnerable to sticky tarpits, at least three of which can be exploited to halt critical operations of the botnet. Our findings thus have the potential to take down botnets, assist in future operations against botnets or slow down botnet activities, and thereby to significantly reduce their economic gain and spreading speed.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments. Furthermore this project has received funding from

the European Union's Horizon 2020 research and innovation program under grant agreement No 700176 ("SISSDEN")

APPENDIX

A. SHA256 Hashes of Samples in Evaluation

The following list details the SHA256 hashes of the exact malware executables we used in our evaluation. The samples can be obtained from VirusTotal or upon request.

- **Bashlite**: self-compiled from https://github.com/anthonygtellez/BASHLITE/blob/master/client.c to sample with SHA256 hash: 525659e0540289d0620d681fea4453a0a9a76da806347e07f10b6a5576868d05
- **Pushdo**: 2a30d7b76e3dcc10861526f83fb060a12485a974626bea8872cf2a012e25333a
- **Pynamer**: 96dc746b7367a0c3ab7cc5d22ac69dc71edef87eff5f80a4aab14c02f37e1bcc
- **SalityP2P**: 69093bb3dd267b8e738320ebe4c954c902636b3298b14771b5da0b0e19c866b2
- **Tofsee**: 031b2f7a46ba809cc1750a10481ff90de900d3e875926784a37e9866926e26b2
- **WannaCry**: 9428b3dc2192473e1be1e155a978f83f8cedbf3f13dbc9b7641806cc0456a81a

REFERENCES

[1] "Revealing Packed Malware, author=Yan, Wei and Zhang, Zheng and Ansari, Nirwan, journal=IEEE Security & Privacy, year=2008, publisher=IEEE."

[2] (anonymous), "BASHLITE," https://github.com/anthonygtellez/BASHLITE, accessed 2018-07-26.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the Mirai Botnet," in *USENIX Security Symposium*, 2017.

[4] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware." in *USENIX Security Symposium*, 2012.

[5] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," in *RAID*, 2007.

[6] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient Detection of Split Personalities in Malware." in *NDSS*, 2010.

[7] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to Recognize Functions in Binary Code." USENIX, 2014.

[8] R. Bhat, "Tracking the Footprints of PushDo Trojan," https://www.blueliv.com/blog-news/research/tracking-the-footproints-of-pushdo-trojan/, accessed 2018-08-07.

[9] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically Dissecting Malicious Binaries," Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, Tech. Rep., 2007.

[10] G. Bruneau and R. Wanner, "DNS Sinkhole," *Reading Room Site. The SANS Institute*, 2010.

[11] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated Test Generation for Worst-Case Complexity," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[12] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The Commoditization of Malware Distribution." in *USENIX Security Symposium*, 2011.

[13] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, 2008.

[14] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities," in *IEEE CSF*, 2009.

[15] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *USENIX HotDep*, 2009.

[16] F. Cybersecurity, "Pushdo It To Me One More Time ," https://www.fidelissecurity.com/sites/default/files/FTA_1016_Pushdo.pdf, accessed 2018-08-07.

[17] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *ACM CCS*, 2008.

[18] T. Eggendorfer, "Reducing Spam to 20% of its Original Value With a SMTP Tar Pit Simulator," in *In MIT Spam Conference*, 2007.

[19] N. Falliere, "All-in-One Malware: An Overview of Sality," https://www.symantec.com/connect/blogs/all-one-malware-overview-sality, accessed 2018-08-06.

[20] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. Van Doorn, "Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking," in *ACM SIGOPS Operating Systems Review*, 2008.

[21] S. Gallagher, "Double-Dip Internet-of-Things Botnet Attack Felt Across the Internet," https://arstechnica.com/information-technology/2016/10/double-dip-internet-of-things-botnet-attack-felt-across-the-internet/, accessed 2018-03-14.

[22] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis *et al.*, "Manufacturing Compromise: The Emergence of Exploit-as-a-Service," in *ACM CCS*, 2012.

[23] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection," in *USENIX Security Symposium*, 2008.

[24] G. Gu, P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee, "Bothunter: Detecting Malware Infection through IDS-Driven Dialog Correlation." in *USENIX Security Symposium*, 2007.

[25] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *NDSS*, 2008.

[26] Hex-Rays, "IDA F.L.I.R.T. Technology: In-Depth," https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml, accessed 2018-10-11.

[27] T. Hunter, P. Terry, and A. Judge, "Distributed Tarpitting: Impeding Spam Across Multiple Servers." in *LISA*, 2003.

[28] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "JACKSTRAWS: Picking Command and Control Connections from Bot Traffic." in *USENIX Security Symposium*, 2011.

[29] J. Jedynak, "A Deeper Look at Tofsee Modules," https://www.cert.pl/en/news/single/a-deeper-look-at-tofsee-modules/, accessed 2018-07-10.

[30] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, 1976.

[31] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-Based Spyware Detection." in *USENIX Security Symposium*, 2006.

[32] J. Kirk, "Microsoft Leads Seizure of Zeus-Related Cybercrime Servers," https://www.computerworld.com/article/2502089/security0/microsoft-leads-seizure-of-zeus-related-cybercrime-servers.html, accessed 2018-05-24.

[33] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host." in *USENIX Security Symposium*, 2009.

[34] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using System-Centric Models for Malware Protection," in *ACM CCS*, 2010.

[35] T. Liston, "Tom Liston Talks about LaBrea," http://labrea.sourceforge.net/Intro-History.html, accessed 2018-03-10.

[36] Microsoft, "__stdcall," https://msdn.microsoft.com/de-de/library/zxk0tw93.aspx, accessed 2018-06-02.

[37] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts," in *IEEE S&P*, 2017.

[38] D. Moore, "Network Telescopes: Observing Small or Distant Security Events," in *Proceedings of the 11th USENIX security symposium*, 2002.

[39] D. Moore, C. Shannon, G. M. Voelker, and S. Savage, *Network Telescopes: Technical Report*, 2004.

[40] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *IEEE S&P*, 2007.

[41] ——, "Limits of Static Analysis for Malware Detection," in *ACSAC*, 2007.

[42] MSDN, "How to Modify the TCP/IP Maximum Retransmission Time-out," https://support.microsoft.com/en-us/help/170359/how-to-modify-the-tcp-ip-maximum-retransmission-time-out, accessed 2018-06-27.

[43] ——, "Send Function," https://docs.microsoft.com/en-gb/windows/desktop/api/winsock2/nf-winsock2-send, accessed 2018-06-27.

[44] ——, "SendBufferSize Property," https://msdn.microsoft.com/en-gb/en-en/library/system.net.sockets.socket.sendbuffersize(v=vs.110).aspx, accessed 2018-06-27.

[45] ——, "Socket Overlapped I/O versus Blocking/Nonblocking Mode," https://support.microsoft.com/en-us/help/181611/socket-overlapped-i-o-versus-blocking-nonblocking-mode, accessed 2018-03-25.

[46] C. Nunnery, G. Sinclair, and B. B. Kang, "Tumbling Down the Rabbit Hole: Exploring the Idiosyncrasies of Botmaster Systems in a Multi-Tier Botnet Infrastructure." in *LEET*, 2010.

[47] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla, "A Comprehensive Measurement Study of Domain Generating Malware." in *USENIX Security Symposium*, 2016.

[48] Reuters, "Shadow Brokers Threaten to Release Windows 10 Hacking Tools," https://tribune.com.pk/story/1423609/shadow-brokers-threaten-release-windows-10-hacking-tools/, accessed 2018-08-07.

[49] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "SoK: P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *IEEE S&P*, 2013.

[50] C. Rossow and C. J. Dietrich, "Provex: Detecting Botnets with Encrypted Command and Control Channels," in *DIMVA*, 2013.

[51] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen, "Prudent Practices for Designing Malware Experiments: Status Quo and Outlook," in *IEEE Security & Privacy*, 2012.

[52] G. Saito and G. Stringhini, "Master of Puppets: Analyzing And Attacking A Botnet For Fun And Profit," *CoRR*, 2015.

[53] B. Stock, J. Göbel, M. Engelberth, F. C. Freiling, and T. Holz, "Walowdac - Analysis of a Peer-to-Peer Botnet," in *Proceedings of the 2009 European Conference on Computer Network Defense*, 2009.

[54] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your Botnet is my Botnet: Analysis of a Botnet Takeover," in *ACM CCS*, 2009.

[55] Unprotect Project, "Sandbox evasion," http://unprotect.tdgt.org/index.php/Sandbox_Evasion, accessed 2018-05-27.

[56] G. Vigna, "When Malware is Packing Heat," https://www.lastline.com/labsblog/malware-packing/, accessed 2018-10-12.

[57] VirusTotal, "YARA," https://github.com/VirusTotal/yara, accessed 2018-10-11.

[58] T. Werner, "Botnet Shutdown Success Story: How Kaspersky Lab Disabled the Hlux/Kelihos Botnet," https://securelist.com/botnet-shutdown-success-story-how-kaspersky-lab-disabled-the-hluxkelihos-botnet-15/31058/, accessed 2018-05-24.

[59] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis using CWSandbox," in *IEEE Security & Privacy*, 2007.

[60] Xtables, "Xtables-Addons," xtables-addons.sf.net, accessed 2018-08-06.

[61] V. Yegneswaran, P. Barford, and D. Plonka, "On the Design and Use of Internet Sinks for Network Abuse Monitoring," in *RAID*, 2004.

[62] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes *et al.*, "SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion," in *RAID*, 2016.