BluePrint: Automatic Malware Signature Generation for Internet Scanning

Kevin Stevens
Georgia Institute of Technology
Atlanta, Georgia, USA
kevin.stevens@gatech.edu

Mert Erdemir
Georgia Institute of Technology
Atlanta, Georgia, USA
merterdemir@gatech.edu

Hang Zhang*
Indiana University Bloomington
Bloomington, Indiana, USA
hz64@iu.edu

Taesoo Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
taesoo@gatech.edu

Paul Pearce
Georgia Institute of Technology
Atlanta, Georgia, USA
pearce@gatech.edu

ABSTRACT

Traditional malware-detection research has focused on techniques for detection on end hosts or passively on networks. In contrast, *global* malware detection on the Internet using active Internet scanning remains relatively unstudied, with research still relying on manual reverse engineering and handwritten scanning code.

In this work, we introduce BluePrint, the first end-to-end system for analyzing new samples of "server-like malware" and automatically preparing and executing Internet scans for them. BluePrint requires only a low degree of human involvement, requiring only analysis of results before launching Internet scans. Important high-level challenges BluePrint must overcome include resiliency and scalability issues with symbolic execution, state explosion from some common networking code patterns, and a high number of duplicate symbolic network signatures with only small, inconsequential structural differences. We solve these with novel binary analysis techniques; respectively, "path sketches" for guided symbolic execution, new symbolic models for `accept()` and `recv()`, and an efficient and effective signature deduplication algorithm.

We evaluate BluePrint on a varied selection of server-like malware, and demonstrate that it successfully identifies infected devices both in simulated local network experiments and on the Internet. We then perform more detailed analyses to characterize the infected hosts found by our scanning experiments, and find that they span a wide range of usage scenarios and geographical locations. We also show that many of these devices seem to have poor general security posture, and that the infections persist for months.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; Software reverse engineering.

*Contributions were partly made while employed at Georgia Institute of Technology.

This work is licensed under a Creative Commons Attribution International 4.0 License.

RAID 2024, September 30–October 02, 2024, Padua, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678923

KEYWORDS

malware, networks, Internet scanning, symbolic execution, automatic protocol reverse engineering

ACM Reference Format:
Kevin Stevens, Mert Erdemir, Hang Zhang, Taesoo Kim, and Paul Pearce. 2024. BluePrint: Automatic Malware Signature Generation for Internet Scanning. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3678890.3678923

1 INTRODUCTION

Malware is a prevalent and increasing threat on the Internet, with hundreds of thousands of new samples discovered each day [96]. 5.5 billion malware attacks were detected in 2022 by SonicWall, up 2% from the previous year [90]. Among others, malware designed for reconaissance or data exfiltration has a high impact. Mandiant has reported that "backdoors," or malware designed for interactive remote use, was the most commonly observed type of malware in 2021, at 40% of all families [70]. "Remote control software" has been similarly reported as the second-most-common type of malware used against Russian government agencies in 2022 [80].

After finding a new malware sample, researchers normally want to understand its behavior, capabilities, and prevalence as quickly as possible. While automated analysis of malware behavior and capabilities is a well-developed field, automatically measuring *prevalence* on the Internet is less-studied. However, understanding the spread of malware is critical for response efforts: notifying law enforcement and owners of infected devices will help them direct efforts to contain and eliminate threats.

Unfortunately, researchers' options for studying the spread of malware are limited. Antivirus providers have the ability to deploy file signatures to their customers and use telemetry data to count infections, but their large customer bases are a resource that most researchers do not have access to. Some early studies conducted measurements of botnets by actively infiltrating their networks [36, 49, 91], but such activities require significant manual effort, and some forms of infiltration may raise legal issues [24].

For families of malware that listen for incoming connections, however, a more accessible approach requiring only a fast Internet connection and a cooperative Internet service provider is possible: scanning for live instances by connecting to and briefly interacting

with remote hosts (*e.g.*, all IPv4 addresses) to identify them. However, despite the existence of mature high-performance scanning tools [26, 27], configuring them for a new malware sample has until now been a basically manual process, requiring a researcher to reverse-engineer the malware and develop a custom scanner module/plugin for it. This both hinders scalability and lengthens the delay between malware detection and scanning. To our best knowledge, no existing work automates this process. In this paper, we aim to explore the extent to which it can be accomplished with automatic binary analysis techniques.

Our Contribution. We develop BluePrint, a system to aid analysts by quickly generating "network signatures" for network-listening, server-like malware. BluePrint then leverages these signatures to execute whole-Internet scans for the malware.

Key challenges include identifying code path(s) likely to implement network handshakes, using them to extract a small number of signatures likely to be scannable, and doing it all efficiently enough to maintain practical throughput, especially including avoiding the well-known problem of state explosion. To solve these, we design and implement techniques such as "path sketches" for symbolic execution guidance, novel symbolic models for key networking APIs, and efficient hash-based signature deduplication.

We evaluate BluePrint on 7 different types of server-like malware (§6), and show that its automatically generated network signatures are sufficient, in most cases, for identifying infections on a test network (§6.4). We then deploy BluePrint and its automatically generated signatures to conduct whole-Internet scans for the malware families, finding real infections (§6.4), and perform population studies (§6.5). Overall, our results demonstrate that BluePrint is an effective tool to aid analysts in rapidly producing network signatures of, and scanning for, server-like malware infections.

2 BACKGROUND

We provide necessary background related to Internet scanning, server-like malware, and its identification in this section.

2.1 Internet Scanning

The introduction of *ZMap*, a fast Internet scanning tool, in 2013 has enabled researchers to answer a wide range of Internet-security-related questions and understand IPv4 topology at a large scale [27]. Being able to scan the entire IPv4 address space in minutes has helped monitor and characterize network attacks [30, 71] as well as detect new security vulnerabilities [102], create new security-oriented solutions [22, 103], understand censorship better [54, 100], and also explore the efficacy of vulnerability notifications [64].

Since most of these studies were initially based on ad-hoc, problem-specific solutions, in-depth analysis was very challenging and laborsome [26]. As a result, efforts to make IPv4 measurement more efficient produced public search engines such as Censys [26] and Shodan [74], which separated active host-discovery-driven Internet scans from IPv4 security analysis by providing queryable *snapshots* of the IPv4 address space, featuring a diverse set of Internet protocols, services, and ports. The introduction of these search engines has also led to the development of a new set of configurable scanning tools such as *ZGrab2* that help researchers gain insights on activity at the application layer of network protocols effectively and efficiently [26]. While established Internet scanning tools best support IPv4 due to its smaller address space, IPv6 scanning is also an active and growing field of research [98].

Although search engines such as Censys and Shodan are effective for studying well-known protocols, they cannot be used for *novel* or *ad-hoc* protocols such as are implemented by many malware families, since their scanners cannot recognize and "speak" them.

2.2 Server-like Malware

We refer to malware that behaves like a server—listening for network connections and interacting with remote hosts that connect to it—as **server-like malware**.

Listening for connections is required for some types of malware, including proxies—like the *BankShot* example we introduce in §3.2—and peer-to-peer botnets. Other network-enabled malware, such as RATs and spyware, can either actively connect to pre-configured command-and-control domains or IP addresses, or passively listen for connections from them, at the author's option. The latter design is less common, because it requires attackers to know *a priori* who their victims are, and for victim devices' ability to listen for external connections to not be blocked by security systems or NATs. However, benefits can include reducing the risk of detection [63, 93], complicating attribution [45, 86], and enabling connections outside of scheduled "phone-home" times [2]. As such, this style is somewhat more common in high-quality, professional malware [5].

Although server-like malware is generally scannable, determining *how* to interact with an arbitrary server to identify whether it is the malware or some other application has until now required manual reverse-engineering and development effort.

2.3 POSIX Socket APIs

To effectively scan for server-like malware, it is essential to understand their communication protocols, *i.e.*, the formats of accepted and response packets. BluePrint automatically extracts these formats from malware binaries by symbolically analyzing program paths processing the packets. To recognize these paths, BluePrint uses standard POSIX socket APIs as anchor points.

As a representative example, Figure 1a illustrates the socket API sequence used by the *BankShot* malware from §3.2. The malware creates a network socket, binds to it, and monitors it for incoming connections. Upon accepting one, it receives the incoming data, processes it, and sends outbound packets in reply. BluePrint is designed to be robust against some variations in this process (§4.1).

3 OVERVIEW

In this section, we provide a high-level overview of our approach to automatic scanning of server-like malware. We begin with definitions of common terms used throughout the paper (§3.1), then illustrate the workflow of BluePrint with an end-to-end example featuring *BankShot*, a real-world malware sample (§3.2).

3.1 Definitions

Packet. In most of the paper, this term refers to a contiguous portion of a TCP data stream. When discussing socket API calls (`recv()` and `send()`), it refers to the input or output of one call.

Challenge Packet. The first packet sent from a remote host—such as a malicious command-and-control (C&C, also commonly abbreviated in the literature as "C2") application, or a network scanning tool—to the server-like malware after connecting to it.

Response Packet. The first packet sent from the malware back to the remote host in response to the challenge packet.

Generation Signature. The format description of a challenge packet recognized by the malware, which would trigger a response packet. The scanner uses it to generate packets to perform a scan.

Validation Signature. The format description of a response packet in reply to a challenge packet following a given generation signature. The scanner uses it to validate whether a response packet is correct.

Signature. A pair of generation and validation signatures, describing a class of expected challenge packets and corresponding response packets for a server-like malware sample. A handshake is the simplest signature, but a malware may have multiple signatures since other interactions can be used as well—*e.g.*, multi-step interactions with the connection simply closed after the first round-trip.

We note that this active scanning-oriented definition is very different from that used by network intrusion detection systems (NIDS), *i.e.*, a pattern passively matched against seen network traffic.

3.2 The Workflow of BluePrint

In this section, we use a real-world malware sample—*BankShot*—to demonstrate at a high level how BluePrint can automatically scan for instances of a server-like malware, given only its binary and a small amount of user guidance as input.

Overview of BankShot. *BankShot* is a "proxy malware" used for data exfiltration and attributed by the US Department of Homeland Security and the Federal Bureau of Investigation to the government of North Korea [19]. By infecting a device that has access to both the public Internet and a private corporate network, a malicious operator can use this proxy as an intermediary to browse the private network and collect confidential information—a technique not unique to *BankShot* [38, 48, 53, 69, 87]. Similar malware has also been used to create illegitimate commercial proxy services [7, 14, 55–58].

BankShot uses a custom communication protocol. An example exchange is shown in Figure 1. To help evade packet-filter-based detection, its initial configuration traffic is obscured with an XOR cipher, with a random seed value prepended to each message.

After establishing a new connection, *BankShot* can recognize several enciphered commands related to preparing the proxy. One of these is the "ping" or "heartbeat" command 0x12348E, to which it simply responds with "ack" command 0x123484.

A key observation is that, in principle, *any* remote host can connect to *BankShot* and positively identify it by checking if it responds correctly to a "ping" command. This makes it possible to perform *active network scans* for the malware by attempting this interaction with many hosts on the Internet. BluePrint automates this, including both analysis and scanning, as described below.

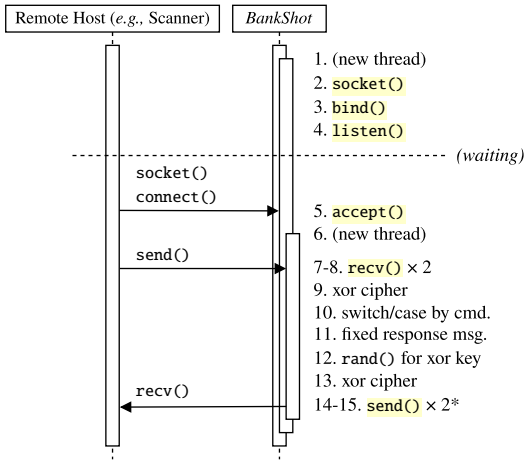
(1) Identifying Packet Processing Paths. Given the *BankShot* binary, our first step is to recognize program paths that may be responsible for receiving and validating challenge packets, and generating and sending responses. To achieve this, BluePrint connects POSIX socket API calls, as mentioned in §2.3. BluePrint overcomes several challenges in this process, such as incomplete socket API sets and paths that may be too short, which we describe in depth in §4.1. Eventually, BluePrint successfully recovers a set of candidate paths, including the one illustrated in Figure 1.

(2) Signature Extraction. With the paths identified above, BluePrint proceeds to extract network signatures using under-constrained symbolic execution. More specifically, it tries to collect all constraints imposed upon the inbound (*i.e.*, received with `recv()`) and outbound (*i.e.*, sent by `send()`) packet data buffers. Although conceptually simpler, we avoid concrete execution (emulation/sandboxing) in our design, because it would suffer from coverage issues in terms of both code paths covered and concreteness of values (*e.g.*, if the malware's response involves random numbers, BluePrint must be able to recognize *any* random numbers as valid responses, not just those that were chosen one time in a simulated environment).

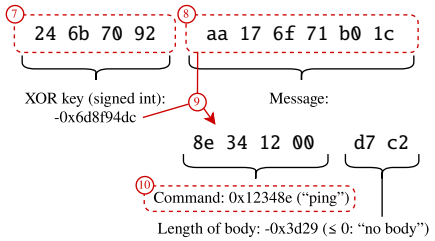
In the case of *BankShot*, as shown in Figure 1, a valid challenge packet must include an XOR cipher key followed by the enciphered (with the key) data; furthermore, to trigger a response packet, the deciphered data must begin with the specific value 0x12348E. The response packet must begin with a randomly generated XOR cipher key as well, followed by the enciphered plain byte sequence `84 34 12 00 00 00`. BluePrint automatically deduces these packet formats, including these complex and subtle constraints, efficiently (*e.g.*, minimizing the potential for state explosion commonly seen in symbolic execution), and encodes them into symbolic signatures. We cover more details of signature extraction in §4.2. In total, BluePrint extracts 12 signatures from *BankShot*.

To select a specific signature to scan with, the extracted signatures are next presented to the user for manual inspection. Approval by an expert analyst is needed before any scanning can begin, as not only would it be wasteful to launch a full Internet scan using a false-positive signature, but some signatures could possibly trigger undesirable behaviors in the malware. This is the only manual step in the entire process, and is unavoidable for safety and ethics reasons. Besides correctness (the malware actually behaves according to the signature), the analyst needs to select a signature that is both safe (does not trigger any malicious behavior) and distinctive (is unlike any known protocols implemented by other software). For *BankShot*, 8 of the 12 signatures are equivalent and correct (our deduplication algorithm, described in §4.2, strikes a balance between speed and completeness, and so does not always catch all duplicates), any of which can be selected.

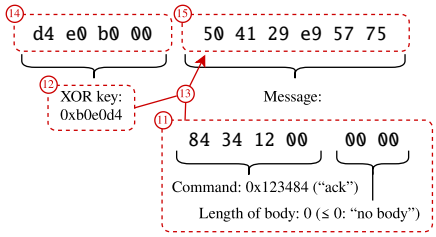
(3) Internet Scanning. The symbolic signature extracted in the previous step enables accurate Internet scanning for live *BankShot* instances. BluePrint first uses the generation signature to generate a set of challenge packets that can elicit recognizable response packets from the malware. For instance, one challenge packet we generate for *BankShot* is `D3 A4 D9 50 5D 71 6F 71 49 96`, where the first four bytes are an XOR cipher key that can be used to decipher the remaining bytes to `8E 34 12 00 2E E1`, of which the leading four bytes match the magic value required to trigger a response packet. BluePrint then performs Internet scanning using the well-established tools *ZMap* [27] and *ZGrab2* [26], with a custom *ZGrab2* module developed by us. We essentially probe Internet hosts with the generated challenge packets, and match any



(a) Operations performed by *BankShot*, and its interaction with a remote host. *BankShot*'s calls to socket APIs are highlighted. *The local TCP/IP stack buffers and combines the two `send()` calls, allowing them to be received by the remote host with a single `recv()`.



(b) Example challenge data (operations 7-10 in Figure 1a). The data shown here was produced automatically by BluePrint.



(c) Example response data (operations 11-15 in Figure 1a).

Figure 1: Overview of a "ping" (0x12348E) interaction with *BankShot*, and the packet data exchanged.

response packets against the previously extracted validation signature (*e.g.*, for *BankShot*, we use the first four bytes to XOR-decipher the remaining bytes and look for the expected plain byte sequence). A successful match indicates a live malware instance.

4 DESIGN

We show the architecture of BluePrint in Figure 2. In this section, we detail the technical challenges BluePrint must overcome and our design choices for each phase of its workflow.

4.1 Packet Processing Path Identification

As mentioned in §3.2, we use symbolic execution to extract packet signatures (data buffer constraints). Rather than executing all program paths—prohibitively slow, since most code is unrelated to the signature—it is essential to provide more detailed guidance to the symbolic executor to reduce its exploration space.

The form of this guidance needs careful consideration, however. If too constrained, valid signatures encoded in alternative paths could be missed. Furthermore, the guidance cannot be expressed as a simple sequence of basic blocks, as it would be inaccurate due to loops and other conditional branches that cannot be precisely predicted statically. To strike a balance between performance and soundness, we introduce the notion of a **path sketch**, which refers to a sequence of nested function calls that we expect to encompass a signature-encoding path. A path sketch provides a valuable yet relaxed hint to the symbolic executor regarding path exploration: on one hand, there is a concrete goal of exploring a path "connecting" all the function calls in the sketch, avoiding the blind walk of an unguided symbolic execution, and on the other hand, the executor has the freedom to explore any path fulfilling the sketch, maximizing the possibility of obtaining valid packet signatures.

Formally, let F be the set of all function addresses in the binary, C the set of all intra-binary function calls (each represented as a pair, $\langle call\_addr, target\_addr \rangle$), and S the set of all addresses of calls to `send()`. A path sketch is a triple, $\langle start\_addr, calls, end\_addr \rangle$, where $start\_addr \in F$, $calls$ is a sequence of elements from C, and $end\_addr \in S$. In a well-formed path sketch, each $call\_addr$ lies within the function pointed to by the previous call's $target\_addr$, and the start and end addresses lie in the same functions as the first call's address and last call's target, respectively.

In our *BankShot* motivating example, the correct path sketch for the scannable network signature starts at `WinMain()` (0x4026F0), includes five function calls, and ends at a call to `send()` at 0x40124E.

Intuitively, to extract signatures for server-like malware, we can construct path sketches by finding and chaining POSIX socket APIs in the malware binary, as mentioned in §2.3. We detail practical challenges with this approach in the remainder of this section.

Incomplete API Set. Ideally, a signature-encoding path sketch should include all the standard socket APIs appearing in a typical network packet-receiving or -sending session, as described in §2.3. However, it is often hard or impossible to recover such a complete API sequence at the binary level: static analysis often has difficulties accurately resolving indirect calls, and dynamic analysis fundamentally suffers from low code coverage. To maximize the likelihood of finding correct sketches, BluePrint just looks for paths that pass through a call to `accept()`, or any ancestor thereof, and end at a call to `send()`. This is valid because a call to `accept()` implies the presence of earlier calls like `listen()` (according to POSIX socket API semantics), which are also less important for packet constraint collection. `recv()` is also not required, as some malware sends data immediately, without waiting for a challenge packet.

Initial Constraints. If a path sketch were to begin at a call to `accept()`, we could miss essential constraints that should be

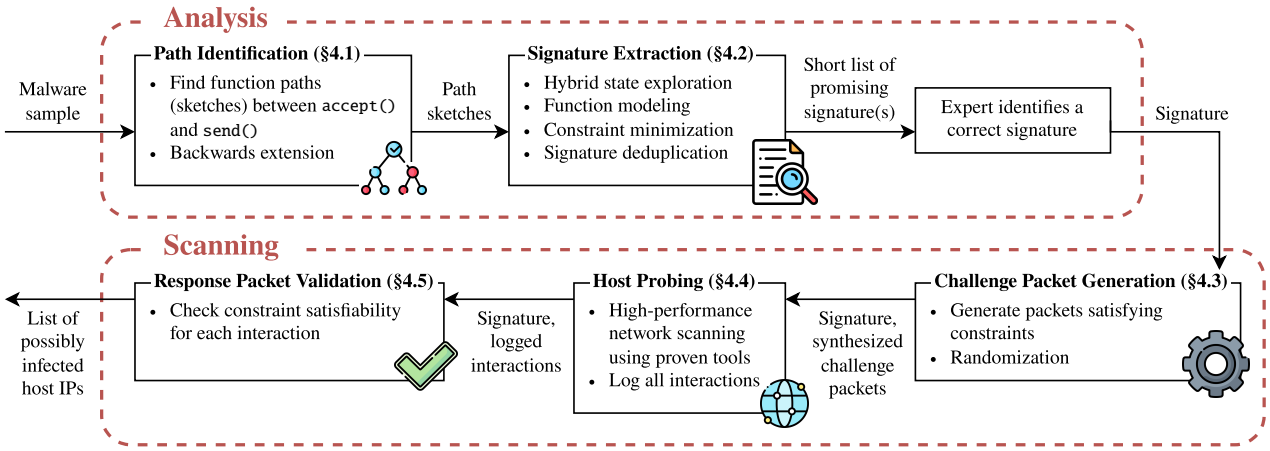


Figure 2: Overview of the BluePrint system architecture.

formed before that point (*e.g.*, initialization of important data structures). We therefore extend each sketch *backward* through direct caller functions, up to the highest we can detect. If there are multiple callers, some sketches may be partially duplicated.

The full path sketch construction algorithm, designed around these considerations, is presented in Appendix A.

4.2 Signature Extraction

Using the path sketches constructed in the last phase, BluePrint proceeds to symbolically execute the malware binary for signature extraction (*i.e.*, collect the constraints of the sent and received packets). In this section, we describe how BluePrint can efficiently perform guided symbolic execution and reliably extract signatures, overcoming various technical challenges.

Hybrid State Exploration. As mentioned in §4.1, we guide the symbolic executor with path sketches for more efficient signature extraction. However, even with this guidance, the symbolic executor still needs to explore feasible paths fulfilling the sketch. To accelerate this, BluePrint employs a *hybrid* state exploration strategy (of breadth-first and depth-first search) to decide which successor basic blocks to symbolically execute after the current one.

More specifically, a path sketch will naturally be divided into **path segments**, each of which refers to all possible paths between two consecutive function call sites in the sketch. We use the path segment as the basic unit of our symbolic execution; at each step, the executor tries to find feasible paths within one segment. To do this, BluePrint first utilizes breadth-first search (BFS) for state exploration, the default strategy employed by widely used symbolic executors such as *angr* [94]. The advantage of BFS is that it explores multiple paths evenly, increasing the chance of hitting a short, feasible one. However, if within a configurable time limit (6 minutes by default) no feasible paths can be found for a particular path segment, BluePrint switches to depth-first search (DFS) until another configurable timeout (4 minutes by default) expires. The intuition is that BFS likely times out because there are too many paths to explore, with the executor making only shallow progress on each—leading to DFS hopefully finding a feasible path more efficiently. This unique hybrid strategy helps BluePrint find feasible

```
1 int recv_all(
2       int sockfd, void *buf, size_t size, int flags) {
3     int total = 0;
4     while (total < size) {
5         int received = recv(
6             sockfd, &buf[total], size - total, flags);
7         if (received <= 0) break;
8         total += received; }
9     return total; }
```

Figure 3: An example of a calling `recv()` in a loop to reach a target amount of data.

paths satisfying the sketch more effectively than either strategy alone. We show more details in §6.3.

Function Modeling. A common practice in symbolic execution is to model frequently invoked functions (*e.g.*, library functions) with handwritten symbolic summaries, avoiding expensive computation and potential state explosion for complex functions. To implement more accurate and efficient symbolic execution, BluePrint models common library functions such as `memcpy()` (inlined or otherwise—more details can be found in §5) in a standard way, and some socket APIs in unique ways specific to our goal. We describe our special socket API modeling below.

(1) `recv()`. The application invokes `recv()` to request a specific amount (via a *size* parameter) of packet data, and it will indicate the actual amount obtained as the return value. This could be less than the requested size, or −1 indicating an error. A straightforward symbolic modeling of `recv()` will thus return a symbolic value between −1 and *size*, potentially leading to a large number of symbolic states if the application conditionally branches on this value. To make matters worse, `recv()` is often invoked in a loop to iteratively receive a target amount of data (a typical loop is shown in Figure 3), frequently causing state explosion in practice.

To address this issue, we design a new symbolic model of `recv()` that avoids state explosion in the common scenario of looped `recv()` invocations. We present our model in Algorithm 1. The first time `recv()` is invoked from a given calling context, the model fills the symbolic buffer with all *size* requested bytes, enabling a quick exit of the wrapping loop shown in Figure 3 and preventing it from

producing state explosion. For further `recv()` invocations from the same calling context, if any, our model returns 0, because in this situation it is likely that the wrapping loop will only terminate when `recv()` returns 0 (*i.e.*, no more data is available).

In addition, after `recv()` returns 0, the socket is marked via metadata in the symbolic execution state to never return any more symbolic packet data in the future, regardless of calling context. This is because making `recv()` return data after previously returning 0 would require the scanner to send more data timed *between* the two calls—a timing window impossible to hit reliably over the Internet, making such a signature effectively a false positive.

Algorithm 1 State-explosion-resistant symbolic `recv()`

```
function RECV(sockfd, buf, size, flags)
    ▷ E_r (exhausted sockets) and C_r (previously seen call
      stacks) are sets stored in the symbolic execution state.
    stack ← GET_CURRENT_CALL_STACK( )
    if sockfd ∈ E_r then                    ▷ Socket is already exhausted
        return 0
    else if stack ∉ C_r then                ▷ First invoc. from this call stack
        add stack to C_r
        WRITE_SYMBOLIC_BYTES(buf, size)
        return size
    else                                    ▷ Later invocations from the same call stack
        add sockfd to E_r
        return 0
```

(2) `accept()`. Similar to `recv()`, `accept()` is frequently called in a loop, in this case to support accepting multiple connections. To prevent this from causing state explosion, we model `accept()` with Algorithm 2. It is structurally similar to Algorithm 1, but omits the no longer applicable "exhausted socket" condition, and behaves differently in the remaining two cases.

Our goal here is to simulate the scenario of scanning, in which the malware receives and accepts exactly one connection. On a real machine, a second call to `accept()` in a loop would block forever if no second connection ever arrived. To model this, we have `accept()` kill the symbolic state if invoked twice from the same calling context. The intuition is that since the second call to `accept()` blocks forever, any code following it is unreachable, *i.e.*, the state is dead at that point. We show that our function modeling can significantly improve symbolic execution efficiency in §6.3.

Algorithm 2 State-explosion-resistant symbolic `accept()`

```
function ACCEPT(sockfd, addr, addrlen)
    ▷ C_a (previously seen call stacks) is a set stored in the
      symbolic execution state.
    stack ← GET_CURRENT_CALL_STACK( )
    if stack ∉ C_a then                     ▷ First invoc. from this call stack
        add stack to C_a
        return CREATE_NEW_UNIQUE_SOCKET_FD( )
    else                                    ▷ Later invocations from the same call stack
        KILL_SYMBOLIC_STATE( )              ▷ Equiv. to blocking forever
```

Constraint Minimization for Signatures. For network scanning, it suffices to include only symbolic constraints relevant to the received and sent packets. However, by default, the symbolic execution engine will record and include constraints for *all* variables along the executed paths—relevant to the packet data or not—in the symbolic states, incurring extra uncertainties and costs for the SMT-solver-based packet generation and validation in the next phases (§4.3). To solve this, we filter out those constraints which are not relevant to the packet data. It is crucial to retain constraints that are *transitively* relevant: for example, "$5 \leq x < 10$" is transitively relevant if a constraint "packet bytes 8–11 are equal to x" exists. We thus build the signature using a standard worklist algorithm, adding constraints involving variables from the worklist to the signature, and those constraints' other variables to the worklist, until empty.

Transitive constraint identification captures an extra 71 constraints in the validation signature for *Soul*, one of our evaluation samples (see §6). These are mostly constraints on a time-and-date struct (*e.g.*, "$1 \leq month \leq 12$") and symbolic format strings.

Signature Deduplication. As the symbolic executor explores feasible paths under the guidance of the path sketches, we collect packet signatures associated with different paths. However, the path differences may not translate to signature differences (*e.g.*, an "if" statement which occurs on the packet-generation path but does not affect the packet itself). As a result, there could be many duplicated signatures (in some cases, over an order of magnitude more than unique signatures, as shown in Table 2) that should be eliminated. To efficiently deduplicate, BluePrint uses a recursive, graph-based hash algorithm over SMT files. Essentially, the algorithm syntactically compares the SMT expressions of packet signatures to decide whether they are identical, in a way resilient to trivial differences such as variable renaming and rearrangement of commutative constraints (*e.g.*, `a && b` and `b && a` are equivalent). The full detailed algorithm is presented in Appendix B.

Although we may miss some duplicates with this algorithm (*e.g.*, semantically equivalent expressions with different syntax structures), we never wrongly identify any duplicates—we intentionally trade comprehensiveness (*e.g.*, could be achieved by a more accurate but also expensive SMT-solver-based solution) for performance.

Signature Format Considerations. We choose the original SMT expression format to encode the packet signatures, instead of transforming them to another widely used packet representation for network scanners, such as the *Snort* rules commonly used in NIDS for malicious packet recognition, or Perl-Compatible Regular Expressions (PCRE) that are also popular in many scanners. Although those representations would be compatible with existing scanning infrastructures, they are mostly designed for simple pattern-matching-based packet recognition, and are far less expressive than SMT expressions. Signatures in those formats are traditionally hand-written, with an analyst picking out a few features they believe to be distinctive and which can be easily pattern-matched for. In contrast, our automatic system extracts the *full* constraints of binary packet data, which can be complex (*e.g.*, bit-level operations, possibly some light cryptography as in *BankShot*).

4.3 Challenge Packet Generation

After an analyst selects one of the symbolic signatures extracted in the previous step (§4.2), BluePrint next generates concrete challenge packets for scanning, by using an SMT solver to find solutions

```
00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 80 15 82 84 00 00 00 00 02 04 10 02 02 00 80
-----------------------------------------------
98 5e 68 90 67 a1 97 6f bd d0 20 31 bf 20 1f fc
2e 09 f7 26 ef 60 fc f1 9e 46 c7 7d 9b 80 e8 0c
33 cf 59 f4 6c 75 4e 8e c5 c9 65 31 05 d9 0e da
```

Figure 4: Top: the first 48 bytes of the 500th packet generated from *Derusbi*'s signature (one of our evaluation samples from §6) by blocking evaluations. Bottom: the same, but using randomization instead. Not pictured is the similarity between packets in each set: with blocking evaluations, the 501st packet differs from the 500th in only one bit, compared to 1015 (of 2048) when using randomization.

to the constraints. BluePrint by default generates 1000 challenge packets, to balance between scan robustness and performance. Each remote host is sent one packet at random during the scan.

An important question is *how* to generate multiple packets with the SMT solver. Although we could configure it to simply generate unique solutions (*e.g.*, by blocking evaluations [8]), the outputs tend to have low randomness, both individually and with respect to each other. We instead configure the SMT solver to generate randomized solutions. We find that this also happens to be faster in most cases, because blocking evaluations accumulates constraints over time (*i.e.*, the next packet must be unequal to all of the previous packets). An example of the difference in output is shown in Figure 4.

4.4 Host Probing

As mentioned in §3.2, BluePrint relies on *ZMap* and *ZGrab2*, which are high-performance, research-oriented Internet scanning tools, for host probing. To support our workflow, we extend the default toolchain with a customized module that allows us to specify the packets to send to the remote hosts and efficiently log the responses for offline processing in the next step.

Separation from Generation and Validation. Since Internet-wide scanning has high performance requirements (*i.e.*, there are millions of remote hosts to probe in the full IPv4 address space), which our SMT-based packet generation and validation cannot meet in general, regardless of our optimizations (see Table 3 from the Evaluation section), we perform those steps offline, separate from the host probing process. Specifically, challenge packets are generated in advance (§4.3) so that the scanner can send them immediately, and the scanner also only records the response packets, to be validated afterwards (§4.5).

4.5 Response Packet Validation

After receiving response packets in reply to our sent challenge packets, we verify whether they match the corresponding validation signature extracted from the malware binary. To do this, BluePrint checks the feasibility of the conjunct constraints of the validation signature and the concrete packet: for example, if a data field f is symbolically constrained to $f < 8$ in the signature and set to a concrete value of 2 in the actual response, we check whether $f < 8 \wedge f == 2$ is satisfiable. If so, we mark the response as from a malware instance. To improve the performance of validating multiple response packets, the symbolic signature constraints are shared in the SMT solver stack, and we only "push/pop" the constraints imposed by different concrete response packets.

One additional case BluePrint supports is the validation of "cross-packet" constraints. For instance, the response packet of *BadCall* malware is only valid if it contains a value from a list provided in the *challenge* packet. Thus, validation may rely on both packets. BluePrint supports this by adding concrete constraints for both packets' SMT variables during validation.

5 IMPLEMENTATION

In this section, we discuss specifics on how we implemented a prototype of BluePrint. Specifics on software versions and technologies used (*e.g.*, IDA Pro) can be found in Appendix C.

Handling Packed Binaries. To evade detection or prevent reverse engineering, some malware authors employ obfuscation such as encrypting or packing their binary code or dynamically importing library functions at runtime. While BluePrint cannot directly analyze such code, obfuscation can often be defeated by simply running such samples in a sandbox and dumping their memory (*e.g.*, with *Process Dump* [75] or *Un{i}packer* [81]) or by using specialized unpackers (*e.g.*, *Magicmida* [42]). We use *Process Dump* to resolve dynamic library imports for two of our evaluation samples in §6, *BankShot* and *FASTCash*, before analyzing them with BluePrint.

Library Function Modeling. As mentioned in §4.2, BluePrint models standard library functions (*e.g.*, `memset()`) in its symbolic execution for efficiency. Our modeling is built upon *angr*'s existing models ("SimProcedures"), and we make several extensions.

First, we support modeling *statically linked* library functions, by recognizing them with *IDA Pro*'s FLIRT signatures [44] and mapping them to SimProcedures. Second, BluePrint can also recognize and model certain *inlined* library functions by pattern-matching their binary instruction sequences. Third, for binaries that dynamically import system DLLs (*e.g.*, with `LoadLibrary()`), BluePrint can use *angr*'s SimProcedures and function signature data to synthesize symbolic DLL models "on the fly." All of these allow BluePrint to apply *angr*'s symbolic function models in more situations, reducing the risk of state explosion from, *e.g.*, `strlen()` or `memcpy()`.

Consecutive send() Invocations. Our Path Identification step aims to find paths that reach any call to `send()`. However, some binaries use multiple consecutive calls to `send()` to respond instead of just one—an implementation detail on their part, since the host's TCP stack can transparently combine such messages. Incorporating this extra data into the signature could improve precision.

To do this, after completing the final path segment, we execute an additional "segment" with no target address, allowing the symbolic execution engine to engage in *unguided exploration* for a short time. Restrictions on returning from functions, which normally prevent backtracking to earlier path segments, are also lifted here.

Handling Multi-Threaded Code in Symbolic Execution. It is common to use multiple threads in network-related code (*e.g.*, to handle simultaneous connections and perform asynchronous background tasks), but precisely reasoning about multi-threaded code in symbolic execution is a well-known difficult problem. Our observation, though, is that connection- and background-task-handling

threads are generally independent from each other, as well as from their host threads. In other words, there is little inter-thread interaction that we need to precisely model. Based on this observation, when symbolically executing multi-threaded code, BluePrint models thread creation (*e.g.*, the Windows `CreateThread()` function) with a two-way state split, in which one state treats it as a blocking function call, and the other as a no-op.

Under-Constrained Indirect Function Calls. We find that in our under-constrained symbolic execution, indirect function calls often have unconstrained target addresses (*e.g.*, virtual method calls on objects for which the constructor was not included in the path sketch). BluePrint is able to skip such calls, using *IDA Pro*'s automatically inferred stack-pointer-delta values to "fix up" the stack pointer. This is functionally similar to *angr*'s default strategy for library functions for which it has no model.

Scan Responses with Low Variability. We find that some samples have only a small number of possible unique responses, despite complex symbolic constraints. (In this case, these constraints relate to *which* packet is sent, rather than to their structures.) As an optimization for validation, if a response signature can only match up to (by default) 16 different packets, we enumerate them and skip invoking the SMT solver for non-matching responses, and also memoize solver results. (We do still need the solver, in order to correctly handle cross-packet constraints (§4.5).)

Randomness. A preliminary scan for *Derusbi* found fewer results than we expected based on our domain knowledge of the malware and its prevalence in the wild. Upon examination, we find that different versions of *Derusbi* fill padding bytes using the `rand()` function in different ways. As this is not a core aspect of the signature (which is primarily about specific negation and rotation operations), we adjust our modeling of `rand()` to return an unconstrained 32-bit integer[1], allowing BluePrint to generate a signature that can match multiple types of `rand()` usage.

6 EVALUATION

Our evaluation aims to answer the following research questions:

- *RQ1.* How accurate are the extracted signatures? Do they faithfully encode the formats and constraints of the challenge and response packets (§6.1)?
- *RQ2.* How efficiently can BluePrint extract signatures from malware binaries (§6.2)?
- *RQ3.* How important is each key technique used by BluePrint? What are their impacts on effectiveness and performance (§6.3)?
- *RQ4.* Can BluePrint identify live malware instances on the Internet (§6.4)?

Setup. Signature extraction and packet validation were performed on a machine with two Intel Xeon E5-2687W v4 processors at 3.0 GHz and 256 GB RAM, running Ubuntu 20.04. Scanning was done on a machine with an Intel Xeon X3470 at 2.93GHz and 24 GB RAM, running Ubuntu 22.04, and with a gigabit Internet connection. Path identification, signature deduplication, and packet generation were executed on a machine with an Intel Core i9-11950H at 2.60GHz and 64 GB RAM, running Ubuntu 22.04.

Dataset. To comprehensively evaluate BluePrint, we assemble a collection of 7 real-world server-like malware samples with various characteristics, listed in Table 1. We select these samples by systematically studying available technical reports on malware (*e.g.*, [19–21, 50, 79, 83]), as well as popular malware families, and then selecting those meeting the criteria (mainly related to implementation-level limitations) listed in §8. As shown in Table 1 and Appendix D, our collected malware samples communicate on different network ports and for different purposes (*e.g.*, to receive commands from C&C clients, for proxying, *etc.*), each with unique challenge and response packet formats. We believe that such a diverse dataset can reliably test BluePrint's capability of automated signature extraction and scanning for a wide range of server-like malware.

6.1 Signature Accuracy

We take a few different approaches to answer *RQ1* in our evaluation. First, we manually reverse engineer all of the malware samples in our dataset to figure out the formats and constraints of their expected challenge and response packets. (These findings are summarized in Table 1, and described in more detail in Appendix E.) Based on this understanding, we then manually verify the correctness of our automatically extracted signatures. Where possible, we do this by directly checking the semantics of the SMT expressions. For signatures too complex to be verified in this way, we use alternative techniques, such as generating sample challenge packets from our signatures and seeing whether they are accepted by the simulated malware packet parsing code, or manually creating sample response packets and seeing whether our validation signatures match them. Furthermore, we also perform scans in a local simulated network environment to ensure that we can detect live malware instances with the extracted signatures (§6.4), proving their effectiveness. With all these approaches, we conclude that BluePrint accurately extracts signatures for all samples in our collection, except for a few minor cases discussed later in this section.

In the rest of this section, we discuss technical challenges presented by our evaluation samples and how BluePrint overcomes them to generate accurate signatures, as well as observed inaccuracies (aforementioned "minor cases") and how they can be addressed.

Complicated Bit-Level Constraints. As mentioned in §3.2, packet formats often involve complex bit-level constraints that have previously required manual effort to deduce. For example, we show two samples' challenge packet validation logic in Figure 5. *Derusbi* uses bit negation and rotation relations between specific integers, and *FASTCash* poses even more complex constraints between certain bytes whose offsets are derived from another non-fixed value. These would be difficult or impossible to express with pattern- or regex-based rules as used in passive NIDS (§4.2).

BluePrint correctly recognizes all these subtle constraints and encodes them into the generated signatures. It achieves this through its use of in-depth symbolic execution to systematically analyze the malware binary, capable of reasoning about fine-grained bit-level operations such as these.

Signature Variants Resulting from Different Paths. Some samples, such as *Gh0st* and *BadCall*, have more than one valid signature

[1]The C language standard leaves `RAND_MAX`, the maximum return value of `rand()`, implementation-defined. For example, it is 0x7FFF in Microsoft's Universal C Runtime Library, but 0x7FFF_FFFF in glibc. By default, *angr* uses 0x7FFF_FFFF on all platforms, including Windows; we instead make `rand()`'s return value entirely unconstrained.

Sample	Type	Conn. Listening Purpose	Handshake (Distinctive Signature) *(see also Appendix E)*	
			Challenge	Response
❶ *BadCall*	Proxy	Proxying	Fake TLS ClientHello	Fake TLS ServerHello, Certificate, and ServerHelloDone
❷ *BankShot*	Proxy	Proxying	XOR cipher seed and ciphered six-byte message	XOR cipher seed and ciphered six-byte message
❸ *Derusbi*	RAT	C&C	Packet with three ints with "magic" relationship	Packet with three ints with "magic" relationship
❹ *FASTCash*	RAT	C&C	Fake TLS packet with two ints with "magic" relationship	Fake TLS packet with two ints with "magic" relationship
❺ *Gh0st*	RAT	Proxying	SOCKS5 handshake: first byte `05`, third `00` or `02`	SOCKS5 handshake: `05 00` or `05 02`
❻ *Slingshot*	Loader	Payload retrieval	None/ignored	`B2 7F 23 43`
❼ *Soul*	RAT	C&C	None/ignored	Fixed HTTP GET header with compressed payload

Table 1: Summary of collected samples. More details are provided in Appendix D.

Sample	Num Skch	Signature Extraction				Gen[2] Time	Ablation			
		Time/ Skch		Dedup			P	H	S	F
		Avg	Max	Count	Time					
❶ *BadCall*	100	15:39	21:25	368 → 23	0:45	5:29	∅	∅	∅	∅
❷ *BankShot*	65	9:13	13:20	36 → 12	<0:01	0:21	∅			∅
❸ *Derusbi*	1	0:03	0:03	1 → 1	<0:01	0:01	∅			
❹ *FASTCash*	75	18:43	36:18	5429 → 16	3:05	0:08	∅		∅	✗
❺ *Gh0st*	159	9:34	22:49	31 → 3	<0:01	27:05	∅		✗	
❻ *Slingshot*	6	18:52	25:05	6 → 1	<0:01	-	∅			
❼ *Soul*	63	35:59	1:05:57	2 → 2	<0:01	-	∅			∅

Table 2: Results of BluePrint's automatic analysis of each sample, and our ablation study (§6.3). Times are mm:ss or h:mm:ss. Ablation: P = path-sketch guidance for symbolic execution, H = hybrid exploration, S = `recv()` and `accept()` state explosion avoidance, F = static and inline function modeling. "∅" = no signatures can be exported when the technique is disabled, "✗" = disabling the technique reduces the quality of the signature (see §6.3 for details).

```
uint32_t *packet = (...);
bool success = (packet[1] == ~packet[0])  // ">>>" =
  && (packet[2] == packet[0] >>> 7);      // right rot.
```

(a) *Derusbi*

```
uint8_t *packet = (...);
uint16_t offset = *(uint16_t*)(&packet[0x15]);
offset = 5 + ~offset;  // 5 bytes for fake
                       // TLSCiphertext header
uint32_t X = ((uint32_t*)(&packet[offset]))[0];
uint32_t Y = ((uint32_t*)(&packet[offset]))[1];
bool success = (((~X ^ 0x3CADEED) << 6) + 0x18472735)
  ^ 0xFC0A397F == Y;
```

(b) *FASTCash*

Figure 5: Key challenge packet validation logic from selected malware samples. (Code refactored for brevity and clarity.)

associated with different packet processing paths. BluePrint therefore supports using the disjunction of multiple signatures when validating scan results. For robust scanning, it is important that it extract *all* possible validation signatures, as missing any could cause false negatives in identifying malware responses. BluePrint reliably achieves this by allowing the symbolic executor to explore different paths fulfilling each path sketch, striking a good balance between performance and comprehensiveness (§4.1).

Discussion of Inaccuracies. We next summarize and discuss the reasons behind BluePrint's inaccuracies when handling some minor cases, as well as our current workarounds and the potential fixes that we will pursue in the future.

(1) Unsupported code features in the implementation. We observed inaccuracies from code features unsupported by our prototype.

First, *BadCall* has a special "abortive shutdown" mechanism [76] in its code, after the `send()` call. Specifically, some code paths will send the response packet *before* the challenge packet has been fully validated. Failure of the postponed verification will cause abortive shutdown, preventing the response packet from actually being sent out in practice. BluePrint cannot capture these additional constraints for the challenge packet beyond the `send()` site, since our path sketches currently end with `send()` (§4.1). This leads to the generation of challenge packets that cannot trigger the response.

Second, the primary network component of *Derusbi* is not implemented with standard POSIX socket APIs (§2.3), making it unanalyzable by our prototype (§4.1). Instead, we analyze the POSIX API-based secondary component which receives packets relayed from the primary one, but with a difference in the expected maximum packet size (256 vs. 64 bytes).

To fix these inaccuracies, we currently manually "patch" the signatures or packets generated by BluePrint, which generally only requires trivial effort (*e.g.*, simply replace "256" with "64"). In the future, we plan to further improve the implementation of BluePrint to support these code features (*e.g.*, post-`send()` verification and non-POSIX socket APIs), which we do not consider as fundamental design limitations of BluePrint.

(2) Inherent difficulties of symbolic execution. It is well known that symbolic execution can have difficulties analyzing complex cryptographic- or compression-related code, whose sophisticated logic can easily lead to state explosion. We identify one such case in *Soul*, which applies DEFLATE compression to the response's HTTP body. As a result, BluePrint has to employ concretization strategies in analyzing such code, causing inaccuracies.

Fortunately, BluePrint can still precisely extract the signature of *Soul*'s HTTP header, which constitutes most of the packet and is by itself already distinctive enough for accurate scanning. Thus, as a workaround, we limit packet validation to the first 95% of the validation signature (about 1160 bytes, essentially corresponding to the header). We believe this problem can be solved generally with better modeling of specific compression algorithms, or improved concretization strategies, which we leave as future work.

[2] 1000 packets were generated from the most correct signature. Samples labeled "-" do not need challenge packets to invoke responses (see Table 1).

6.2 Efficiency

To answer *RQ2*, we summarize the time cost of BluePrint when processing our malware collection in Table 2. Path sketches are constructed within seconds for all samples, and there are typically fewer than 80 (max: 159) unique sketches per sample. BluePrint then spends on average around 20 minutes per path sketch on symbolic execution, in a single thread—parallelism would be straightforward because each sketch is executed independently. Our deduplication algorithm can efficiently (within minutes) reduce the number of signatures per malware sample from up to thousands to fewer than 25. Packet generation from a specific signature can also be finished in minutes. As will be shown later in Table 3, BluePrint can also validate the response packets in a reasonable amount of time for real-world scanning.

The only significant manual step in the BluePrint pipeline is signature selection. As mentioned in §3.2, this is an ethical requirement as much as a technical one, as the signature and binary *must* be inspected by a human to ensure that a scan will not trigger any malicious behavior. BluePrint provides the path sketch from which each signature came in a simple format, making it easy for an analyst to check it in the binary. The signature itself is sometimes human-readable, but to help with complex cases, BluePrint also provides tools to generate and validate example packet data. Ultimately, the speed of this step varies based on the complexity of the sample and its extracted signatures, and the experience level of the analyst. Samples such as *Slingshot* can be studied and greenlit in a matter of minutes, whereas malware with relatively complex signature-related behavior such as *Soul* will likely take several hours. We note that this is much faster and easier than fully reverse-engineering the malware without the help of BluePrint.

6.3 Effectiveness of Analysis Techniques

To answer *RQ3* and gain a deep understanding of how each key technique (as described in §4) benefits signature extraction, we conduct an ablation evaluation for each of them and summarize the results in Table 2. We discuss the impact of each technique below.

Path-Sketch-Guided Symbolic Execution. We disable the use of path sketches to guide symbolic execution (§4.1), and instead run unguided symbolic exploration from pre-specified entry points. After 1 hour of execution with the BFS path exploration strategy and 1 hour with DFS, the results show that no signatures can be extracted, since the relevant packet processing code has not been explored by the symbolic executor. This highlights the importance of path sketch guidance for efficient signature extraction.

Hybrid Path Exploration Strategy. To assess the impact of our hybrid path exploration strategy (§4.2), we replace it with *angr*'s default BFS strategy and rerun the symbolic execution. In this setting, we find that the symbolic execution cannot finish for *BadCall* within the time limit, due to its complex packet validation and generation logic. Our hybrid strategy, however, can employ DFS to finish analyzing some feasible packet processing paths in time.

recv() and accept() State Explosion Avoidance. We disable our state-explosion-avoidance algorithms (§4.2) for `recv()` and `accept()`—keeping `recv()`'s packet lengths symbolic, and removing our "block forever" / "kill the symbolic state" condition for `accept()`—and rerun the symbolic execution.

State explosion prevents us from extracting signatures from *BadCall* and *FASTCash* in this case. Due to a hard-coded limit on symbolic packet lengths in *angr*, we initially also find that the length of *Gh0st*'s generation signature is reduced from 0x5000 bytes to 256. Although the shorter signature would still work correctly for *Gh0st*, it would not necessarily work in other, similar situations. We disabled the limit and tested again, and found that no signatures were produced this time, due to the higher constraint complexity.

Function Modeling. We disable BluePrint's detection and modeling of static and inlined library functions (§5), leaving only DLL functions modeled, and rerun the symbolic execution. We find that the resulting state explosion prevents BluePrint from finding any signatures for three of our 7 samples.

For a fourth sample, *FASTCash*, the state explosion from a particular `memcpy()` loop "splits" the signature into many different versions, one for each possible length of the response packet (which the malware randomizes). While BluePrint supports validation using the disjunction of multiple signatures, as mentioned in §6.1, the range of possible packet lengths in this case leads to several *hundred* separate signatures being produced, and in fact most of them cannot even be exported before a timeout expires. We thus find that BluePrint's function modeling techniques are key.

6.4 Network Scans

To answer *RQ4* and verify whether the signatures extracted by BluePrint can locate live malware instances, we perform network scans both in a local simulated environment and on the real-world Internet. We detail our network scan evaluation in this section.

Setup. For the local simulated scans, we use two *VirtualBox* virtual machines connected by a virtual internal network. One runs Windows 11 and is used to host malware instances in a sandboxed environment (*e.g.*, to simulate a victim host), and the other runs Debian 11 Bullseye and is used to perform the scan.

We use the same machine as in the previous evaluations to compare the response packets to validation signatures (§6). The specifications of the machine used for the real-world Internet scans are in §6. We validate the response packets in 48 parallel processes.

Local Scan Results. To verify the effectiveness of our extracted signatures, we first run scans in a controlled local-network setting with two connected virtual machines. In these experiments, we successfully detect all of the live malware instances in our collection, except for *Derusbi* and *Slingshot*, whose execution environments are difficult to prepare in a virtual machine and we thus skipped. Nevertheless, the accuracy of our signatures for those two samples is manually verified as described in §6.1. This demonstrates the reliability of the signatures generated by BluePrint, and makes a solid preparation for the subsequent Internet scans.

Internet Scan Results. After validating the effectiveness of our signature generation and scanning tools locally, we conduct a series of whole-IPv4-space scans using BluePrint. Table 3 shows a summary of those results, whereby we find tens of real-world advanced persistent threat (APT) infections. There are two core challenges associated with conducting these scans: expectations and ports.

We note that broadly, we would not expect to find significant real-world infections. This is due to both that our samples are

Sample	Port	Date	Validation Time	Num. Results	Num. Confirmed
❶ *BadCall*	80	Nov. 2023	26:28:35	0	0
	443	Nov. 2023	26:24	0	0
	8000	Dec. 2023	42:28	0	0
❷ *BankShot*	80	Dec. 2023	55:50:41	0	0
	110	Dec. 2023	5:50:18	0	0
	443	Dec. 2023	15:38:14	0	0
❸ *Derusbi*	80	Apr. 2023	16:43	14	14
	443	Apr. 2023	5:52	7	7
❹ *FASTCash*	80	Nov. 2023	16:09:21	0	0
	443	Nov. 2023	2:32:39	0	0
❺ *Gh0st*	80	Apr. 2023	5:54	2,174	0
	443	Apr. 2023	4:19	1,461	0
	1080	Dec. 2023	1:43:55	55,649	0
❻ *Slingshot*	80	Apr. 2023	2:53	0	0
	443	Apr. 2023	1:00	0	0
❼ *Soul*	80	Apr. 2023	2:33:21	0	0
	443	Apr. 2023	57:55	0	0

Table 3: Internet scan results. *Slingshot* and *Soul* do not need challenge packets, so we reused *Derusbi*'s scan data for them.

of well-known malware that has already been analyzed by the community and with mitigations deployed, and the relatively short lifespan of most malware campaigns [89]. Any infections we find are likely remnants of prior campaigns, or abandoned installations.

Malware may run on any port, and it is prohibitive to exhaustively scan all IPv4 addresses *and* ports. Thus, we must select a set number of ports to explore, while weighing the cost of Internet scans vs. the likelihood of success. We observed in prior analysis that for server-like malware, port 80 and 443 were often used, given their likelihood to be open by firewalls [79]. Thus, for each sample, we scan on ports 80 and 443, plus any default ports (Appendix D).

BluePrint successfully identifies multiple active infections of the APT malware *Derusbi*, demonstrating its potential. We describe this population of infections more subsequently (§6.5). We envision BluePrint deployed as a framework to continuously analyze emerging malware and monitor network hosts, significantly improving the community's responsiveness to network malware.

False Positives. Our tool also identified a population of potential *Gh0st* infections, that upon inspection, are actually SOCKS5 proxies. This is due to the sample leveraging the SOCKS5 proxy protocol [61]. Although the signature BluePrint extracts is correct, a signature matching a standardized protocol like this naturally risks false-alarm scan results, as it may match other, non-malicious applications. We therefore manually identify some unusual implementation details of *Gh0st*'s SOCKS5 handshake, and confirm these hosts to indeed be SOCKS5 proxies and not *Gh0st* infections.

6.5 *Derusbi* Population Studies

In this section, we conduct a series of population studies for the *Derusbi*-positive hosts from Table 3. We aim to characterize these hosts and understand their continued activity after almost a year.

Country	# of ASes	# of /24 Subnets	# of *Derusbi*-Positive IPs
South Korea	3	4	4
Sweden	1	1	3
India, Vietnam (each)	1	1	2
Italy, Taiwan, USA (each)	1	1	1
Total	**9**	**10**	**14**

Table 4: The country, autonomous system (AS), and /24 subnet breakdown of *Derusbi*-positive hosts. The most common country is South Korea with 3 unique ASes and 4 hosts, followed by Sweden with 3 hosts under one /24 network.

Locations and Autonomous Systems (ASes). We show the breakdown of the countries, ASes, and /24 subnets for the 14 *Derusbi*-positive hosts in Table 4. We see that four of the hosts are located in South Korea, each belonging to a separate /24 subnet, two of which are in the same AS. Sweden, with one AS, is the second most common country, accounting for three hosts.

We use ASdb [104] to map ASes to organizations. We found that 6 of the ASes are in the Internet Service Providers (ISPs) class, two of which are also Hosting/Cloud Providers. Separately, two ASes are classified under Education & Research, being operated by a university and a science institute. This infection pattern aligns with the authors' experiences regarding this malware.

Reverse DNS (PTR) Lookups. To see if any of the hosts are mapped to a domain name, we performed reverse DNS lookups to get their PTR records. Using ZDNS [47] on Google's Public DNS server [39], we successfully mapped nine hosts to at least one domain name. One IP address returned four records for different versions of a website for summer internship programs at a science institute in South Korea. The other Education & Research-category host has a record for a language program at a university in Taiwan. Four addresses have records indicating they may be used in infrastructure serving static content for three ASes. Lastly, three addresses under a single /24 mapped to domain names that, we believe, indicate they may be used as Web Feature Service servers. These last three domain names serve the *Microsoft IIS 7* default landing page on port 80, supporting this.

Longitudinal Activeness. As of March 2024, slightly less than a year since the original scanning experiments, 10 out of 14 (six out of seven for port 443) *Derusbi*-positive hosts remain active (*i.e.*, respond to pings), though one of them resets connections on both ports 80 and 443. Six hosts remain infected (five for port 443). Therefore, our longitudinal experiment shows that the majority of infected hosts remain accessible (71.4% for port 80, 85.7% for port 443) roughly a year later, and of those, the majority remain infected as well (60.0% for port 80, 83.3% for port 443).

Open Ports. To better understand which ports are publicly exposed by 9 still-active *Derusbi*-positive hosts, we performed a *Nmap* [68] scan of the 100 most common ports. Across all hosts, only ports 53, 80 and 443 were found to be open. Eight hosts had both 80 and 443 open, and one had 80 open and 443 active but closed (*i.e.*, no service is listening on it). The remaining two had all three ports open, and one was responsive to a DNS query for `example.com`, indicating

an open DNS resolver. One host also had ports 7, 23, 110, and 8000 active but closed. We should also note that when we crawled one of the IP addresses in India, the webpage, served on port 80 without TLS, appears to provide access to the ISP's historical Telecom Consumer Complaints Monitoring System database for both read and write access (based on function names on the webpage). However, we did not interact with the webpage for the ethical reasons. We believe these port patterns are problematic, considering these types of services should not be publicly exposed.

TLS Logs and Certificates. We used *ZGrab2*'s HTTP module to scan on port 443 the active hosts still infected with *Derusbi*, to obtain TLS and website information in order to better categorize the kinds of hosts compromised by the malware.

The TLS landscape varied significantly, with some being self-signed, some being formerly trusted but having expired years ago, and others being current active certificates. In Sweden, the discovered hosts included a small business, as well as the tourist website for a local municipality. Interestingly, one of the hosts in South Korea was also a tourist website for a local municipality. In addition to the above hosts, we also found *Derusbi* on a website purporting to be a corporation providing air navigation services, as well as on a device displaying a *pfSense* [77] firewall appliance login page.

Conclusion. Our population study of *Derusbi*-positive hosts finds open DNS servers, critical ISP infrastructure and internal services, and firewalls exposed to the public, across a wide range of countries and ASes. In addition, many infected hosts belong to ISPs, universities, and science institutes, making the severity of the infections more sensitive and concerning. The fact that BluePrint can identify infections of well-known malware families for which signatures have long been known indicates the usefulness of the system and the need to develop systems to conduct global Internet scans for malware to identify and remediate infections.

6.6 Ethics

When conducting Internet scans, we followed community norms and best practices [27]. These included randomizing IP addresses to avoid straining individual networks, using IP addresses with WHOIS and DNS records that identify the academic research nature of the networks, hosting a webpage on the scanning IPs and machines with contact information and opt-out instructions, and honoring all opt-out requests.

Scanning for malware, however, presents a specific new set of ethical challenges. We need to ensure that our communication with these devices does not in any way constitute authentication, commands, or any form of action that could harm infected machines. Thus, when selecting network signatures to use for our scanning experiments, we carefully checked the reverse-engineered malware code to ensure that our packets would not trigger any malicious behavior on infected hosts, and we performed local experiments to verify this. We also ensured that the scanning signatures generated did not "complete" authentication, application-level connections, or command behaviors, and were the minimum viable set of packets to identify infection. Development and local scanning experiments were run only on virtual machines with no Internet access.

We have provided our information on *Derusbi* infections to law enforcement.

7 RELATED WORK

Malware Internet Scanning. Although tools like *Nmap* [68], *ZMap* [27] or *ZGrab2* [26] can be used to detect some security vulnerabilities [27, 103], they cannot automatically extract network signatures from malware samples. Population studies of server-like malware are thus often done as part of larger analyses of campaigns (*e.g.*, of botnets [71]), for which reverse engineering is being done anyway. These tools are also often used to find *possibly* infected hosts, by fingerprinting devices or filtering on specific open ports [16, 37, 51]. Services such as Censys [26], Shodan [74], and GreyNoise [40] can be used as historical search engines, but lack customizability and are often slow to provide new labels.

Most malware-related Internet scanning studies focus on either botnets, by mimicking their peer-to-peer traffic [28, 43, 60], or RATs, by mimicking infected machines connecting to C&C servers [72]. Botnet scanners are generally ad-hoc [28, 43], though *ZMap* has also been used [60]. Shodan Malware Hunter continuously scans for RAT C&C servers from a few known families [41, 73]. Narrower RAT scanning studies have investigated state-sponsored campaigns [72], amateur operators [30], and victims [29]. Of studies searching for server-like malware using active Internet scanning, BluePrint is the first to mostly automate network signature extraction.

Automatic Protocol Reverse Engineering. Network signature generation is a form of *automatic protocol reverse engineering*, which can include protocols such as file formats and USB. Tools in this space are often classified by whether they use network traces or execution traces as input [25, 46, 52, 88]. The latter approach, used by BluePrint, has a higher potential for accuracy because the program code serves as ground truth for the logic applied to the protocol data; however, it has lost popularity over time [46], likely due to its invasiveness. Despite this, we believe that the accuracy that can be gained through this method is important for real-world applicability, so we focus here only on studies that use it.

Additionally, we discuss only studies on the recovery of protocol fields and their constraints ("PF recovery"), as research on extracting protocol *finite state machines* ("PFSM recovery") is not useful for Internet scanning, where only the initial interaction is relevant.

Several early works [11, 12, 78] address only the relatively simple "replay problem": identifying certain fields (*e.g.*, hostnames and checksums) that must be edited before a network trace can be replayed. These were followed by research aimed at generating more general descriptions of application input data formats, starting with Polyglot [13] and subsequently improved with techniques such as using multiple sample messages [99], using call stack information [66], careful loop analysis [18], and analyzing control dependencies [67]. A few works, such as FFE/x86 [65] and P2C [59], have studied automatic characterization of application *output* formats. ReFormat [97] is notable for its ability to analyze encrypted protocols by finding plain-text data in memory.

Most of these studies use ad-hoc packet description formats instead of SMT, limiting their expressiveness in favor of human understandability. In addition, none use full symbolic execution with state forking, likely to avoid state explosion. Our novel approach using path sketches (§4.1) creates an effective middle ground between concrete execution and empirically non-scalable full symbolic exploration, though BluePrint could potentially be further

improved with some of these techniques such as improved handling of loops and detection and handling of encrypted protocols.

BluePrint is also, to our best knowledge, the first to implement both generation of input data *and* validation of output data. We believe it important for our program-recognition use case that these tasks be combined in a single tool, as using separate tools for the extraction of input and output formats would lose the temporal/dependency relationships between *corresponding* inputs and outputs, and would be unable to identify or express "cross-packet" constraints that span across them (§4.5).

Automatic Analysis of Malware Network Protocols. In addition to Dispatcher [11], two other works have used automatic techniques to study malware protocols. [3] uses symbolic execution to generate example packets for a RAT, in order to exercise its functionality in a sandbox. Their research is limited to one direction of communication, and does not involve analysis of the malware's responses as would be needed for scanning. [10] increases the effectiveness by introducing additional symbolic execution techniques, but has the same goal as [3] and ignores malware response data.

8 CONCLUDING DISCUSSION

In this work, we introduce BluePrint, the first system able to automatically analyze server-like malware samples and prepare and execute Internet scans for them. Our evaluation shows that BluePrint can successfully analyze and scan for a wide variety of server-like malware, using novel techniques to address challenges such as incomplete path visibility, state explosion, and duplicate network signatures. We show this by using BluePrint to identify 14 real malware infections on the Internet, on devices found to have widely varying purposes and geographic locations.

Limitations. As BluePrint relies on static analysis and symbolic execution, it requires malware that is not obfuscated or packed, or can be dumped from memory in a plain form. It also currently uses standard POSIX socket APIs (§2.3) to construct path sketches, but as shown earlier in §6.1, not all malware implements network functionality with these APIs, making it difficult for BluePrint to analyze them. It cannot create path sketches passing through most indirect calls, as resolving indirect call targets is a difficult problem in static code analysis generally. BluePrint also may not be able to handle signatures involving multiple variable-length TCP response packets, as aligning the symbolic buffers to the concrete TCP datastream (*e.g.*, two calls to `send()` but only one to `recv()`) is a combinatorially difficult problem. It currently only targets x86 Windows malware written in compiled languages like C and C++, but this is constrained only by implementation, not methodology.

No matter how advanced BluePrint or similar systems become, malware using passive connections can be made completely unscannable, *e.g.*, by requiring all requests to be digitally signed with an attacker's key. Since other common analysis techniques like symbolic execution can also be intentionally thwarted [4, 85], but are still useful in many cases, we do not consider this limitation significant. Alternatively, malware could detect scanning by BluePrint, *e.g.*, by providing commands not used during normal operation. Since scans should be conducted openly according to best practices [27], detection itself is not a concern. If malware were to specifically trigger malicious behavior upon detecting scanning, this fact would be spotted by an analyst during signature selection, and the signature rejected like any other malicious command.

Future Work. In principle, BluePrint could support scans for server-like applications in general; however, we have not explored this area in depth. We did investigate applying BluePrint to command-and-control software, but found that most are written in managed languages such as C#, unsupported by our prototype.

Production symbolic execution engines often trade accuracy for performance, using techniques such as symbolic-address concretization [95] and introduction of artificial constraints (*e.g.*, *angr* by default limits the size of memory allocations to 128 bytes) to reduce the amount of symbolic variability being tracked. However, today's symbolic execution engines do not provide any means of tracking when, where, or how this occurs, which in BluePrint's case can lead to generated signatures being overly constrained, potentially limiting their usefulness for scanning in some cases. This issue affected *Soul* as described in §6.1, but similar problems affected other samples during testing of early versions of BluePrint. These were alleviated when we improved our symbolic exploration and function modeling techniques, but nevertheless, we believe this problem to be general, and that it would be worthwhile to develop a technique to reliably track symbolic values affected by these types of heuristics. The user could then be alerted to potential problems with the signature, and affected byte ranges or constraints could be skipped when validating scan responses.

BluePrint could also be extended to scan for multiple similar samples ("variants") simultaneously. A simple approach would be to analyze each one separately, and take the conjunction of each correct signature's challenge-packet constraints and the disjunction of their response-packet constraints. Analysis time could potentially be saved by reusing results across variants, and in cases where no single scan can cover all of the variants, a minimal set of scans could be produced. A more ambitious extension would be to generalize signatures to *unavailable* variants—*e.g.*, "based on the ways in which five given variants of *BankShot* differ from each other, create a signature that should work with *any* similar *BankShot* variants."

Finally, several of our evaluation samples use ad-hoc implementations of established network protocols such as TLS and SOCKS5, with slight deviations from the official specifications. Such signatures can lead to false-positive scan results, as seen with *Gh0st* in §6.4. By adding built-in knowledge of common protocols, BluePrint could warn the user of such similarities, or even automatically tailor generated challenge packets to exercise any deviations from the corresponding protocol specification in order to reduce or eliminate false-positive results.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the ONR under grants N00014-23-1-2080 and N00014-23-1-2095, and by Cisco Systems. Icons in Figure 2 made by Freepik from www.flaticon.com [31–35].

REFERENCES

[1] Christopher Allen and Tim Dierks. 1999. The TLS Protocol Version 1.0. RFC 2246. https://doi.org/10.17487/RFC2246

[2] Eduardo Altares, Joie Salvio, and Roy Tay. 2022. *GoTrim: Go-based Botnet Actively Brute Forces WordPress Websites.* Retrieved January 5, 2024

from https://www.fortinet.com/blog/threat-research/gotrim-go-based-botnet-actively-brute-forces-wordpress-websites

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *Proceedings of the 1st International Symposium on Cyber Security Cryptography and Machine Learning (CSCML '17)*, Shlomi Dolev and Sachin Lodha (Eds.). Springer International Publishing, 171–188. https://doi.org/10.1007/978-3-319-60080-2_12

[4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2991079.2991114

[5] Bradley Barth. 2017. *Researchers link new backdoor and Mac-based rootkit to Turla spy group.* Retrieved January 5, 2024 from https://www.scmagazine.com/news/researchers-link-new-backdoor-and-mac-based-rootkit-to-turla-spy-group

[6] Pete Beck and David Cannings. 2014. *NCC Group Malware Technical Note: Derusbi Server variant (November 2014).* Technical Report. Retrieved July 25, 2023 from https://research.nccgroup.com/wp-content/uploads/episerver-images/assets/800eeb255c4149708f9cc9415f7e2700/800eeb255c4149708f9cc9415f7e2700.pdf

[7] Bitsight Threat Research Team. 2023. *Unveiling Socks5Systemz: The Rise of a New Proxy Service via PrivateLoader and Amadey.* Retrieved January 4, 2024 from https://www.bitsight.com/blog/unveiling-socks5systemz-rise-new-proxy-service-privateloader-and-amadey

[8] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. 2022. *Programming Z3.* Retrieved April 4, 2023 from https://theory.stanford.edu/~nikolaj/programmingz3.html

[9] Simon Blake-Wilson, Jan Mikkelsen, Magnus Nystrom, David Hopwood, and Tim Wright. 2003. Transport Layer Security (TLS) Extensions. RFC 3546. https://doi.org/10.17487/RFC3546

[10] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution. In *Proceedings of the 3rd International Symposium on Cyber Security Cryptography and Machine Learning (CSCML '19)*, Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung (Eds.). Springer International Publishing, 121–140. https://doi.org/10.1007/978-3-030-20951-3_12

[11] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. Association for Computing Machinery, New York, NY, USA, 621–634. https://doi.org/10.1145/1653662.1653737

[12] Juan Caballero and Dawn Song. 2007. *Rosetta: Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NAT Rewriting.* Technical Report CMU-CyLab-07-014. Carnegie Mellon University CyLab. Retrieved December 29, 2023 from https://software.imdea.org/~juanca/papers/cmucylab07014.pdf

[13] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286

[14] Ofer Caspi. 2023. *ProxyNation: The dark nexus between proxy apps and malware.* Retrieved January 4, 2024 from https://cybersecurity.att.com/blogs/labs-research/proxynation-the-dark-nexus-between-proxy-apps-and-malware

[15] Ziv Chang, Kenney Lu, Aaron Luo, Cedric Pernet, and Jay Yaneza. 2015. *Operation Iron Tiger: Exploring Chinese Cyber-Espionage Attacks on United States Defense Contractors.* Technical Report. Trend Micro.

[16] Sam Coyle. 2024. Port Scanning Techniques Tools and Detection. *Preprints* (2024). https://doi.org/10.20944/preprints202403.0225.v1

[17] C.Rufus Security Team. 2008. *Gh0st RAT Beta 2.5* 开源-红狼远控. Retrieved June 5, 2023 from https://www.15897.com/blog/post/Gh0st-RAT-Beta-2.5-open-source.html

[18] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820

[19] Cybersecurity & Infrastructure Security Agency (CISA). 2017. *Malware Analysis Report (MAR) - 10135536-B.* Technical Report. Retrieved May 24, 2023 from https://web.archive.org/web/20220529212912/https://www.cisa.gov/uscert/sites/default/files/publications/MAR-10135536-B_WHITE.PDF

[20] Cybersecurity & Infrastructure Security Agency (CISA). 2018. *MAR-10201537 – HIDDEN COBRA FASTCash-Related Malware.* Technical Report. Retrieved May 5, 2023 from https://www.cisa.gov/news-events/analysis-reports/ar18-275a

[21] Cybersecurity & Infrastructure Security Agency (CISA). 2019. *MAR-10135536-10 – North Korean Trojan: BADCALL.* Technical Report. Retrieved May 5, 2023 from https://www.cisa.gov/news-events/analysis-reports/ar19-252a

[22] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 5–17. https://doi.org/10.1145/2810103.2813707

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 11th European Joint Conference on Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08/ETAPS '08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[24] David Dittrich and Erin Kenneally. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research.* Technical Report. U.S. Department of Homeland Security.

[25] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2018. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques* 14, 1 (2018), 53–68. https://doi.org/10.1007/s11416-016-0289-8

[26] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. 2015. A Search Engine Backed by Internet-Wide Scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2810103.2813703

[27] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *Proceedings of the 22nd USENIX Security Symposium (SEC '13)*. USENIX Association, 605–619.

[28] Marc Eisenbarth and Jason Jones. 2013. BladeRunner: Adventures in Tracking Botnets. In *Botnet Fighting Conference 2013 (Botconf '13)*. Alliance internationale de lutte contre les botnets, Rezé, France.

[29] Brown Farinholt, Mohammad Rezaeirad, Damon McCoy, and Kirill Levchenko. 2020. Dark Matter: Uncovering the DarkComet RAT Ecosystem. In *Proceedings of The Web Conference 2020 (WWW '20)*. Association for Computing Machinery, New York, NY, USA, 2109–2120. https://doi.org/10.1145/3366423.3380277

[30] Brown Farinholt, Mohammad Rezaeirad, Paul Pearce, Hitesh Dharmdasani, Haikuo Yin, Stevens Le Blond, Damon McCoy, and Kirill Levchenko. 2017. To Catch a Ratter: Monitoring the Behavior of Amateur DarkComet RAT Operators in the Wild. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP '17)*. 770–787. https://doi.org/10.1109/SP.2017.48

[31] Freepik. 2024. Check. https://www.flaticon.com/free-icon/check_3285799.

[32] Freepik. 2024. Decision tree. https://www.flaticon.com/free-icon/decision-tree_5139787.

[33] Freepik. 2024. File. https://www.flaticon.com/free-icon/file_1150643.

[34] Freepik. 2024. Gear. https://www.flaticon.com/free-icon/gear_1790071.

[35] Freepik. 2024. Worldwide. https://www.flaticon.com/free-icon/worldwide_2859731.

[36] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. 2005. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *Proceedings of the 10th European Conference on Research in Computer Security (ESORICS '05)*. Springer-Verlag, Berlin, Heidelberg, 319–335. https://doi.org/10.1007/11555827_19

[37] Vincent Ghiette, Norbert Blenn, and Christian Doerr. 2016. Remote Identification of Port Scan Toolchains. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMI '16)*. 239–243. https://doi.org/10.1109/NTMS.2016.7792471

[38] Sivagnanam Gn and Sean Gallagher. 2020. *Ransomware operators use SystemBC RAT as off-the-shelf Tor backdoor.* Retrieved January 4, 2024 from https://news.sophos.com/en-us/2020/12/16/systembc/

[39] Google. 2024. Google Public DNS. https://developers.google.com/speed/public-dns.

[40] GreyNoise. 2024. GreyNoise. https://www.greynoise.io/.

[41] Levi Gundert. 2015. *Proactive Threat Identification Neutralizes Remote Access Trojan Efficacy.* Technical Report. Recorded Future. Retrieved April 16, 2024 from https://go.recordedfuture.com/hubfs/reports/threat-identification.pdf

[42] Hendi48. 2024. Magicmida. https://github.com/Hendi48/Magicmida.

[43] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. 2019. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS '19)*. The Internet Society.

[44] Hex-Rays. 2024. F.L.I.R.T. (2024). Retrieved March 28, 2024 from https://hex-rays.com/products/ida/tech/flirt

[45] Vladislav Hrčka. 2023. *WinorDLL64: A backdoor from the vast Lazarus arsenal?* Retrieved January 5, 2024 from https://www.welivesecurity.com/2023/02/23/winordll64-backdoor-vast-lazarus-arsenal/

[46] Yuyao Huang, Hui Shu, Fei Kang, and Yan Guang. 2022. Protocol Reverse-Engineering Methods and Tools: A Survey. *Computer Communications* 182 (2022), 238–254. https://doi.org/10.1016/j.comcom.2021.11.009

[47] Liz Izhikevich, Gautam Akiwate, Briana Berger, Spencer Drakontaidis, Anna Ascheman, Paul Pearce, David Adrian, and Zakir Durumeric. 2022. ZDNS: A Fast DNS Toolkit for Internet Measurement. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22)*. Association for Computing Machinery, New York, NY, USA, 33–43. https://doi.org/10.1145/3517745.3561434

[48] jcleebobgatenet. 2022. *Attackers Using FRP (Fast Reverse Proxy) to Attack Korean Companies.* Retrieved January 4, 2024 from https://asec.ahnlab.com/en/38156/

[49] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. 2008. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1455770.1455774

[50] Kaspersky. 2018. *The Slingshot APT.* Technical Report. Retrieved June 5, 2023 from https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/09133534/The-Slingshot-APT_report_ENG_final.pdf

[51] Hwankuk Kim, Taeun Kim, and Daeil Jang. 2018. An Intelligent Improvement of Internet-Wide Scan Engine for Fast Discovery of Vulnerable IoT Devices. *Symmetry* 10, 5 (2018), 151. https://doi.org/10.3390/sym10050151

[52] Stephan Kleber, Lisa Maile, and Frank Kargl. 2019. Survey of Protocol Reverse Engineering Algorithms: Decomposition of Tools for Static Traffic Analysis. *IEEE Communications Surveys & Tutorials* 21, 1 (2019), 526–561. https://doi.org/10.1109/COMST.2018.2867544

[53] Vyacheslav Kopeytsev and Seongsu Park. 2021. *Lazarus targets defense industry with ThreatNeedle.* Technical Report. Retrieved January 4, 2024 from https://ics-cert.kaspersky.com/media/Kaspersky-ICS-CERT-Lazarus-targets-defense-industry-with-Threatneedle-En.pdf

[54] Lukas Krämer, Johannes Krupp, Daisuke Makita, Tomomi Nishizoe, Takashi Koide, Katsunari Yoshioka, and Christian Rossow. 2015. AmpPot: Monitoring and Defending Against Amplification DDoS Attacks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '15)*. 615–636. https://doi.org/10.1007/978-3-319-26362-5_28

[55] Brian Krebs. 2019. *The Rise of "Bulletproof" Residential Networks.* Retrieved January 4, 2024 from https://krebsonsecurity.com/2019/08/the-rise-of-bulletproof-residential-networks/

[56] Brian Krebs. 2022. *A Deep Dive Into the Residential Proxy Service '911'.* Retrieved January 4, 2024 from https://krebsonsecurity.com/2022/07/a-deep-dive-into-the-residential-proxy-service-911/

[57] Brian Krebs. 2022. *The Link Between AWM Proxy & the Glupteba Botnet.* Retrieved January 4, 2024 from https://krebsonsecurity.com/2022/06/the-link-between-awm-proxy-the-glupteba-botnet/

[58] Brian Krebs. 2023. *Who and What is Behind the Malware Proxy Service SocksEscort?* Retrieved January 4, 2024 from https://krebsonsecurity.com/2023/07/who-and-what-is-behind-the-malware-proxy-service-socksescort/

[59] Yonghwi Kwon, Fei Peng, Dohyeong Kim, Kyungtae Kim, Xiangyu Zhang, and Dongyan Xu. 2015. P2C: Understanding Output Data Files via On-the-Fly Transformation from Producer to Consumer Executions. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*. The Internet Society.

[60] Ricky "HeadlessZeke" Lawshae. 2014. *Hunting Botnets with ZMap.* Retrieved March 29, 2024 from https://web.archive.org/web/20160808012806/http://community.hpe.com/t5/Security-Research/Hunting-Botnets-with-ZMap/ba-p/6320865

[61] Marcus D. Leech. 1996. SOCKS Protocol Version 5. RFC 1928. https://doi.org/10.17487/RFC1928

[62] Marcus D. Leech. 1996. Username/Password Authentication for SOCKS V5. RFC 1929. https://doi.org/10.17487/RFC1929

[63] Brandon Levene, Robert Falcone, and Tyler Halfpop. 2017. *Kazuar: Multiplatform Espionage Backdoor with API Access.* Retrieved January 8, 2024 from https://unit42.paloaltonetworks.com/unit42-kazuar-multiplatform-espionage-backdoor-api-access/

[64] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. 2016. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *Proceedings of the 25th USENIX Security Symposium (SEC '16)*. USENIX Association, 1033–1050.

[65] Junghee Lim, Thomas Reps, and Ben Liblit. 2006. Extracting Output Formats from Executables. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. IEEE Computer Society, 167–178. https://doi.org/10.1109/WCRE.2006.29

[66] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS '08)*. The Internet Society.

[67] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering* 36, 5 (2010), 688–703. https://doi.org/10.1109/TSE.2009.54

[68] Gordon "Fyodor" Lyon. 2009. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* Insecure, Sunnyvale, CA, USA.

[69] Asheer Malhotra, Vitor Ventura, and Jungsoo An. 2022. *Lazarus and the tale of three RATs.* Retrieved January 4, 2024 from https://blog.talosintelligence.com/lazarus-three-rats/

[70] Mandiant. 2022. *M-Trends 2022: Mandiant Special Report.* Technical Report. Retrieved July 27, 2023 from https://www.mandiant.com/resources/reports/m-trends-2022-insights-todays-top-cyber-trends-and-attacks

[71] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium (SEC '17)*. USENIX Association, 1093–1110.

[72] William R. Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. 2014. When Governments Hack Opponents: A Look at Actors and Technology. In *Proceedings of the 23rd USENIX Security Symposium (SEC '14)*. USENIX Association, 511–525.

[73] John Matherly. 2024. Shodan - Malware Hunter. https://malware-hunter.shodan.io.

[74] John Matherly. 2024. Shodan Search Engine. https://www.shodan.io.

[75] Geoff McDonald. 2022. Process Dump. https://github.com/glmcdona/Process-Dump.

[76] Microsoft. 2021. *Graceful Shutdown, Linger Options, and Socket Closure.* Retrieved October 18, 2023 from https://learn.microsoft.com/en-us/windows/win32/winsock/graceful-shutdown-linger-options-and-socket-closure-2

[77] Netgate. 2024. pfSense. https://www.pfsense.org.

[78] James Newsome, David Brumley, Jason Franklin, and Dawn Song. 2006. Replayer: Automatic Protocol Replay by Binary Analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. Association for Computing Machinery, New York, NY, USA, 311–321. https://doi.org/10.1145/1180405.1180444

[79] Novetta. 2014. *Derusbi (Server Variant) Analysis.* Technical Report. Retrieved March 30, 2023 from https://web.archive.org/web/20220119130812/https://www.novetta.com/wp-content/uploads/2014/11/Derusbi.pdf

[80] Positive Technologies. 2023. *Positive Research / 2023.* Technical Report. Retrieved July 27, 2023 from https://www.ptsecurity.com/upload/corporate/ww-en/analytics/positive-research-2023-eng.pdf

[81] Christian Presa Schnell and Samuel Hopstock. 2024. Un{i}packer. https://doi.org/10.5281/zenodo.11236807

[82] RSA. 2014. *RSA Incident Response: Emerging Threat Profile Shell_Crew.* Technical Report. Retrieved June 5, 2023 from https://web.archive.org/web/20210719111000/https://www.rsa.com/content/dam/en/white-paper/rsa-incident-response-emerging-threat-profile-shell-crew.pdf

[83] Rotem Sde-Or. 2022. *The Hunt for the Lost Soul: Unraveling the Evolution of the SoulSearcher Malware.* Retrieved May 3, 2023 from https://www.fortinet.com/blog/threat-research/unraveling-the-evolution-of-the-soul-searcher-malware

[84] Yali Sela. 2015. *Gh0st RAT: What Is It and How do You Find It?* Retrieved April 5, 2023 from https://www.sentinelone.com/blog/the-curious-case-of-gh0st-malware

[85] Toshiki Seto, Akito Monden, Zeynep Yücel, and Yuichiro Kanzaki. 2019. On Preventing Symbolic Execution Attacks by Low Cost Obfuscation. In *Proceedings of the 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '19)*. 495–500. https://doi.org/10.1109/SNPD.2019.8935642

[86] Ryan Sherstobitoff. 2018. *Analyzing Operation GhostSecret: Attack Seeks to Steal Data Worldwide.* Retrieved January 5, 2024 from https://www.mcafee.com/blogs/other-blogs/mcafee-labs/analyzing-operation-ghostsecret-attack-seeks-to-steal-data-worldwide/

[87] Sergei Shevchenko. 2020. *Cloud Snooper Attack Bypasses AWS Security Measures.* Technical Report. Retrieved January 4, 2024 from https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-cloud-snooper-report.pdf

[88] Baraka D. Sija, Young-Hoon Goo, Kyu-Seok Shim, Huru Hasanova, and Myung-Sup Kim. 2018. A Survey of Automatic Protocol Reverse Engineering Approaches, Methods, and Tools on the Inputs and Outputs View. *Security and Communication Networks* 2018 (2018). https://doi.org/10.1155/2018/8370341

[89] Camelia Simoiu, Ali Zand, Kurt Thomas, and Elie Bursztein. 2020. Who is Targeted by Email-Based Phishing and Malware? Measuring Factors That Differentiate Risk. In *Proceedings of the 20th ACM Internet Measurement Conference (IMC '20)*. Association for Computing Machinery, New York, NY, USA, 567–576. https://doi.org/10.1145/3419394.3423617

[90] SonicWall. 2023. *2023 SonicWall Cyber Threat Report.* Technical Report.

[91] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. 2009. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. Association for Computing Machinery, New York, NY, USA, 635–647. https://doi.org/10.1145/1653662.1653738

[92] Symantec Threat Hunter Team. 2021. *New Espionage Campaign Targets South East Asia.* Retrieved June 5, 2023 from https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/espionage-campaign-south-east-asia
[93] Van Ta, Jake Nicastro, Rufus Brown, and Nick Richard. 2021. *FIN13: A Cybercriminal Threat Actor Focused on Mexico.* Retrieved January 5, 2024 from https://www.mandiant.com/resources/blog/fin13-cybercriminal-mexico
[94] The angr Project contributors. 2023. *Simulation Managers - angr documentation.* Retrieved May 25, 2023 from https://docs.angr.io/en/latest/core-concepts/pathgroups.html#exploration-techniques
[95] The angr Project contributors. 2023. *Symbolic memory addressing - angr documentation.* Retrieved July 22, 2023 from https://docs.angr.io/en/latest/advanced-topics/concretization_strategies.html
[96] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. 2019. A Close Look at a Daily Dataset of Malware Samples. *ACM Transactions on Privacy and Security (TOPS)* 22, 1 (2019). https://doi.org/10.1145/3291061
[97] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS '09)*, Michael Backes and Peng Ning (Eds.). Springer, Berlin, Heidelberg, 200–215. https://doi.org/10.1007/978-3-642-04444-1_13
[98] Grant Williams, Mert Erdemir, Amanda Hsu, Shraddha Bhat, Abhishek Bhaskar, Frank Li, and Paul Pearce. 2024. 6Sense: Internet-Wide IPv6 Scanning and its Security Applications. In *Proceedings of the 33rd USENIX Security Symposium (SEC '24)*. USENIX Association.
[99] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. 2008. Automatic Network Protocol Analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS '08)*. The Internet Society.
[100] Eric Wustrow, Colleen M. Swanson, and J. Alex Halderman. 2014. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *Proceedings of the 23rd USENIX Security Symposium (SEC '14)*. USENIX Association, 159–174.
[101] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. 138–157. https://doi.org/10.1109/SP.2016.17
[102] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J. Alex Halderman. 2015. Neither Snow Nor Rain Nor MITM... An Empirical Analysis of Email Delivery Security. In *Proceedings of the 15th ACM Internet Measurement Conference (IMC '15)*. Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/2815675.2815695
[103] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 14th ACM Internet Measurement Conference (IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755
[104] Maya Ziv, Liz Izhikevich, Kimberly Ruth, Katherine Izhikevich, and Zakir Durumeric. 2021. ASdb: A System for Classifying Owners of Autonomous Systems. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC '21)*. Association for Computing Machinery, New York, NY, USA, 703–719. https://doi.org/10.1145/3487552.3487853

A PATH SKETCH CONSTRUCTION ALGORITHM

A function call is defined as a pair: $\langle call\_addr, target\_addr \rangle$. A path sketch is defined as a triple: $\langle start\_addr, calls, end\_addr \rangle$, where *calls* is a sequence of function calls.

---

function CALL_ADDRESSES_TRAVERSED(p: path sketch)
  return $\{c.call\_addr$ **for** c **in** $p.calls\} + \{p.end\_addr\}$

function FUNCTIONS_TRAVERSED(p: path sketch)
  return {GET_CONTAINING_FUNCTION(c)
    for c **in** CALL_ADDRESSES_TRAVERSED(p)}

function ALL_PATH_SKETCHES_TO(c: function call address)
  ▷ Initialize worklist with a single path sketch.
  $W \leftarrow \{\langle$GET_CONTAINING_FUNCTION(c), [], $c\rangle\}$
  ▷ Extend path sketches as far backwards as possible.
  $F \leftarrow \emptyset$ ▷ "Finished" path sketches
  while $W \neq \emptyset$ **do**
    $p \leftarrow$ **remove** arbitrary member **from** W
    $C_n \leftarrow$ ALL_CALL_ADDRESSES_TO($p.start\_addr$) ▷ "New" calls
    $C_e \leftarrow$ CALL_ADDRESSES_TRAVERSED(p) ▷ "Existing" calls
    $b \leftarrow$ false ▷ "Added any new path sketches" flag
    for $c_n \leftarrow$ members of C_n **do**
      ▷ "$c_n \notin C_e$" ensures the algorithm terminates.
      if $c_n \notin C_e \wedge$ IS_WITHIN_A_FUNCTION(c_n) **then**
        add $\langle$GET_CONTAINING_FUNCTION(c_n),
          $[\langle c_n, p.start\_addr\rangle] + p.calls$,
          $p.end\_addr\rangle$ **to** W
        $b \leftarrow$ true
    if not b **then** ▷ No possible extensions were found.
      add p **to** F
  return F

function ALL_PATH_SKETCHES_TO_ANY(C: set of call addresses)
  return $\bigcup_{c \in C}$ ALL_PATH_SKETCHES_TO(c)

function CONSTRUCT_PATH_SKETCHES( )
  ▷ Get set of function addresses to filter by: ancestors of `accept()`.
  $C_a \leftarrow$ ALL_CALL_ADDRESSES_TO("`accept`")
  $P_a \leftarrow$ ALL_PATH_SKETCHES_TO_ANY(C_a)
  $A_a \leftarrow \bigcup_{p \in P_a}$ FUNCTIONS_TRAVERSED(p)
  ▷ Get all path sketches leading to `send()`, and filter them.
  $C_s \leftarrow$ ALL_CALL_ADDRESSES_TO("`send`")
  $P \leftarrow \emptyset$ ▷ Final set of path sketches
  for $p \leftarrow$ members of ALL_PATH_SKETCHES_TO_ANY(C_s) **do**
    ▷ Only include path sketches that pass through ancestors of `accept()`.
    if FUNCTIONS_TRAVERSED(p) $\cap A_a \neq \emptyset$ **then**
      add p **to** P
  return P

B SMT FILE HASHING ALGORITHM FOR DEDUPLICATION

function NORMALIZED_TEXT(t: SMT2 term)
  return t's SMT2 "`(let`" string representation, with all variable names replaced with a placeholder (*e.g.*, "`x`"), and not including the inner term itself

▷ Here, the "children" of a term are any others to which it directly refers (*e.g.*, `(let ((a (and b c))) ...)` has two, `b` and `c`).

▷ Not shown: HASH_TERM should be memoized.
function HASH_TERM(t: SMT2 term)
  if t is "`declare-fun`" type **then** ▷ Base case
    return HASH(⟨t.var_name, t.var_size⟩)
  else if t is commutative **then** ▷ "and," "or," *etc.*
    $H_c \leftarrow$ SUM(HASH_TERM(c) **for** c **in** t.children)
    return HASH(⟨NORMALIZED_TEXT(t), H_c⟩)
  else
    $HL_c \leftarrow$ [HASH_TERM(c) **for** c **in** t.children]
    return HASH(⟨NORMALIZED_TEXT(t), HL_c⟩)

function HASH_SMT_FILE(f: SMT2 file)
  ▷ "Flatten" any nested "and"s or "or"s.
  for $t \leftarrow$ terms in f **do**
    if t is "and" or "or" type **then**
      merge t with all direct children of same type
  return HASH_TERM(f.root) ▷ The innermost term

C SUMMARY OF TECHNOLOGIES USED IN PROTOTYPE IMPLEMENTATION

Component	Runtime	Library or API	LoC
Path Ident.	CPython 3.10.12	*IDA Pro* 7.7	1433
Sig. Ext.	PyPy 7.3.12 (3.10)	*angr* 9.2.54 [101]	3796
Packet Gen.	PyPy 7.3.12 (3.10)	*Z3* 4.10.2.0 [23]	687
Net. Scan.	Go 1.19.4	*ZGrab2* c9a9ac1 [26]	1352
Packet Val.	CPython 3.10.13	*Z3* 4.10.2.0	1289

D ADDITIONAL DETAILS OF EVALUATED MALWARE SAMPLES

"PE Year" is the timestamp embedded in the executable header, which could indicate the compilation date, but may be spoofed. "Lit. Year" is the earliest known reference in the literature, forming an upper bound.

Sample	.text Size	Year (PE; Lit.)	Default Port
❶ *BadCall*	51 KB	2016; 2019 [21]	8000
❷ *BankShot*	439 KB	2016; 2017 [19]	110
❸ *Derusbi*	28 KB	2012; 2014 [82]	All[3]
❹ *FASTCash*[4]	81 KB	2016; 2018 [20]	443
❺ *Gh0st*	379 KB	2017; 2015 [15][5]	1080[6]
❻ *Slingshot*	25 KB	2016; 2018 [50]	443
❼ *Soul*	134 KB	2017; 2021 [92]	Unknown[7]

Sample	SHA-256
❶ *BadCall*	`d1f3b9372a6be9c02430b6e452620297`
	`4179a674ce94fe22028d7212ae6be9e7`
❷ *BankShot*	`780a9da4b933d0eb457f71666a72f596`
	`163b6ef22756e760a7e222e920d0cf4b`
❸ *Derusbi*	`dfb81afc08cd1510319c2a41101bfeb9`
	`872c4ffcce979122018bdf904b654e7d`
❹ *FASTCash*	`ab88f12f0a30b4601dc26dbae57646ef`
	`b77d5c6382fb25522c529437e5428629`
❺ *Gh0st*	`fcda71cec001e447be7bf0120b0fd7c1`
	`84d2c6a58ae32e209721505266f696d8`
❻ *Slingshot*	`fa513c65cded25a7992e2b0ab03c5dd5`
	`c6d0fc2282cd64a1e11a387a3341ce18`
❼ *Soul*	`69a9ab243011f95b0a1611f7d3c333eb`
	`32aee45e74613a6cddf7bcb19f51c8ab`

E NETWORK SIGNATURE DESCRIPTIONS

We describe here only the network signatures that we selected for our scanning experiments, based on distinctiveness and safety (*i.e.*, the interaction does not trigger any malicious behavior). Some of the samples implement additional signatures that could be used.

E.1 *BadCall*

Implements a "fake" TLS 1.0 [1] handshake (*i.e.*, no meaningful information is exchanged, and subsequent communication uses a custom protocol with no relation to TLS). The challenge packet is a ClientHello, and the response is a sequence of ServerHello, Certificate, and ServerHelloDone. The interaction largely follows the specification, but with a few simplifications:

- `TLSPlaintext.fragment.length` is ignored.
- `TLSPlaintext.fragment.body.client_version` is ignored.
- The cipher suite is selected from the client-supplied list at random, without considering its validity.

The "abortive shutdown" behavior described in §6.1 occurs if the challenge packet does not include a Server Name Indication extension [9] with any of 20 names it recognizes.

E.2 *BankShot*

This signature is also described in §3.2 and illustrated in Figure 1.

Challenge. Receives 10 bytes. The first four form an XOR cipher key used to encipher the remaining six. After deciphering, the next four bytes must be `83 34 12 00`, and the last two, read as a signed little-endian integer, indicate an amount of additional data to receive. (Zero or a negative value indicates no additional data.)

[3] *Derusbi* uses a localhost socket to connect to a malicious kernel driver that relays all TCP connections starting with its signature [6, 79, 82].
[4] This RAT is described as "*FASTCash*-related malware" in [20]; we refer to it as "*FASTCash*" for brevity.
[5] *Gh0st* is a well-known and highly varied [84] family of remote-access trojans (RATs) dating back to 2008 [17]. Since there are no references to our specific sample (by hash) in the literature, we broaden our search to include any mentions of the strings "`jingtisanmenxiachuanxiao.vbs`" or "`Game Over Good Luck By Wind`", which only appear in certain *Gh0st* strains similar to ours.
[6] This default port is defined by the C&C application, not in the malware sample itself.
[7] Our sample did not initially include configuration data for passive listening.

Response. Sends 10 bytes. The first four form a randomly selected XOR cipher key used to encipher the remaining six, which decipher to `84 34 12 00 00 00`.

E.3 *Derusbi*

Challenge. Receives 256 bytes (specifically 64 when considering the entire malware platform, as explained in §6.1) satisfying the following condition:

```
uint32_t *packet = (...);
bool success = (packet[1] == ~packet[0])  // ">>>" =
  && (packet[2] == packet[0] >>> 7);      // right rot.
```

Response. Sends 64 bytes satisfying the same condition as above, and otherwise randomized.

E.4 *FASTCash*

Implements a "fake" exchange of TLS 1.0 [1] packets. In contrast to the highly structured TLS handshake implemented by *BadCall* (§E.1), this signature mimics regular encrypted traffic (*i.e.*, not a beginning-of-connection handshake), so most of the challenge data is ignored and most of the response data is randomized.

Challenge. Receives 237–381 bytes (depending on the "offset" value shown below) satisfying the following condition:

```
uint8_t *packet = (...);
uint16_t offset = *(uint16_t*)(&packet[0x15]);
offset = 5 + ~offset;  // 5 bytes for fake
                       // TLSCiphertext header
uint32_t X = ((uint32_t*)(&packet[offset]))[0];
uint32_t Y = ((uint32_t*)(&packet[offset]))[1];
bool success = (((~X ^ 0x3CADEED) << 6) + 0x18472735)
  ^ 0xFC0A397F == Y;
```

Response. Sends 389–900 (randomized length) bytes. The first five form a TLSCiphertext header: `17 03 01`, followed by a two-byte length field. The rest satisfy the condition described above for the challenge packet, and are otherwise randomized.

E.5 *Gh0st*

Implements proxying with a variation of the SOCKS5 protocol [61, 62]. We use the authentication interaction as the signature, to avoid performing actual proxying for safety and ethical reasons.

Challenge. Receives 0x5000 bytes. The first must be `05` (representing SOCKS'5') and the third must be `00` or `02` (requested authentication method: "none" or "username/password").

The "NMETHODS" field specified in [61] (*i.e.*, the second byte) is ignored. This is the primary "unusual implementation detail" we use in our follow-up scan (§6.4) to identify *Gh0st*.

Response. Sends `05 00` or `05 02`, depending only on whether the malware was configured with a SOCKS5 username and password.

This is a second quirk we use in §6.4, as compliant implementations should only choose from authentication methods requested by the client [61]. The third is that authentication method `01` ("GSS-API"), mandated by [61], is not supported (the connection is closed).

E.6 *Slingshot*

Challenge. None required.

Response. Sends four constant bytes, `B2 7F 23 43`.

E.7 *Soul*

This sample's protocol is described in more detail in [83].

Challenge. None required.

Response. Sends an HTTP "GET" request consisting of a highly distinctive fixed 1115-character-long string followed by an accurate Date header timestamp, Content-Length header, and DEFLATE-compressed buffer with information about the infected system.