Hitchhiking Vaccine: Enhancing Botnet Remediation With Remote Code Deployment Reuse

Runze Zhang[1], Mingxuan Yao[1], Haichuan Xu[1], Omar Alrawi[1],
Jeman Park[2]*, Brendan Saltaformaggio[1]*
[1]*Georgia Institute of Technology* [2]*Kyung Hee University*

***Abstract*—For decades, law enforcement and commercial entities have attempted botnet takedowns with mixed success. These efforts, relying on DNS sink-holing or seizing C&C infrastructure, require months of preparation and often omit the cleanup of left-over infected machines. This allows botnet operators to push updates to the bots and re-establish their control. In this paper, we expand the goal of malware takedowns to include the covert and timely removal of *frontend* bots from infected devices. Specifically, this work proposes seizing the malware's built-in update mechanism to distribute crafted remediation payloads. Our research aims to enable this necessary but challenging remediation step after obtaining legal permission. We developed ECHO, an automated malware forensics pipeline that extracts payload deployment routines and generates remediation payloads to disable or remove the frontend bots on infected devices. Our study of 702 Android malware shows that 523 malware can be remediated via ECHO's takedown approach, ranging from covertly warning users about malware infection to uninstalling the malware.**

I. Introduction

Botnet takedowns have limited success in practice, with ample real-world cases of how complex operations can easily fail [1]–[5]. Traditional botnet takedowns rely on blocking and sinkholing C&C servers. To aid this effort, the research community proposed effective techniques for identifying C&C servers [6] and disabling C&C infrastructure through DNS [7] or taking over P2P botnets [8], [9]. Unfortunately, if incident responders miss *even a single* C&C server during the takedown, botnet operators can easily regain control by pushing an update to the frontend bots [10]. The principal limitation allowing botnets to survive takedowns is that *infected devices continue to run frontend bots* and quickly update to resume their operations [2], [11].

Seeking a different solution, this paper proposes that takedown campaigns must not only block backend C&C servers but, more importantly, *disable or remove frontend malware*. This approach puts malware operators at a disadvantage: They must reinfect devices, which is the most resource-demanding task for building a botnet. However, existing solutions fall short of this goal. Paleari et al. [12] proposed analyzing the frontend bots to enumerate artifacts that must be removed during malware remediation. TARDIS [13] and Shan et al. [14] restore clean backups of infected systems to remove malware. Unfortunately, these works rely on all infected device owners to perform the remediation on their devices. This is ineffective, given that device owners lack reliable sources to recognize the infection and deploy remediation solutions. Ideally, remediation should be deployed globally. However, this requires access to all infected devices, making prior solutions unrealistic [4].

Interestingly, the insight to solving this challenge lies in the bot's implementation, namely *remote payload deployment*. Instead of embedding malicious code in the malware, malware fetches on demand the malicious payload hosted on C&C servers [15]. Previous studies [16], [17] have found that malware assumes full trust in their C&C servers. Besides providing rapid code updates, payload distribution can also easily evade static-vetting systems since malicious logic is not present until it is fetched [18], [19]. While beneficial to malware operators, this tactic of modern malware also inspired us: If incident responders can induce a global removal "update," they could inoculate the malware on globally distributed victim devices.

In fact, recent legal developments [20] set a precedent that newly enables this approach (see §VII). Incident responders can now receive legal leverage during a lawful takedown operation to remove the frontend malware from infected devices, effectively eliminating the chances for a botnet revival. Our research aims to formalize and automate this necessary but complicated remediation procedure.

This paper presents ECHO, an automated pipeline for remediating remotely controlled malware by seizing and reusing their payload deployment routines. ECHO takes a malware sample as input. ECHO's overall objective is to reuse the remote payload deployment capabilities enabled by the malware operators to remediate the malware. We first define a fundamental model to represent the remote payload deployment routine in a malware and enable remediation payload generation (§III-A). Based on that model, §III-B will extract all payload deployment routines in the malware that are candidates for remediation payload generation. Next, §III-C determines what influence each remediation payload can have on the victim system when deployed inside the malware. Combining all these findings, §III-D generates a remediation payload template for incident responders.

As a proof of concept, we implemented our ECHO prototype for Android malware. Our prototype handles two classes of payload deployment routines: 1) Remote Dynamic Code Loading (*RDCL*) for Java binaries (JAR) and Android

[*] Co-corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241353
www.ndss-symposium.org

class files (DEX and APK) and 2) JavaScript (JS) payloads via WebView's Javascript Interface (*JSI*). Remote payload deployment is not exclusive to Android malware, and ECHO's methodology can be extended to remediate malware on any platform, as detailed in §VI-B.

ECHO was designed and evaluated in collaboration with Netskope, a leading Secure Access Service Edge (SASE) provider. We evaluated ECHO on 702 Android malware samples that potentially implement remote payload deployment (detailed in §IV). Our analysis revealed that ECHO applies to 523 out of 702 malware samples for remediation, effectively disrupting the malware's execution or notifying the device's user. ECHO identified that 465 out of 580 RDCL malware (80.17%) can be disrupted using a generated Java remediation payload, and JS payloads can be generated for 75 out of 170 (44.12%) malware with JSI payload deployment routines. These findings demonstrate ECHO's potential for integration into proactive malware mitigation platforms, like those provided by Netskope. §V presents two case studies showcasing how ECHO can warn users and instruct them to remove the malware. ECHO is available at https://github.com/CyFI-Lab-Public/ECHO.

II. Motivation

Using the fake Youku malware[1] as an example, we demonstrate how incident responders can rapidly remediate the malware with proposed approach. Our research found seven malware samples that mimic popular Chinese video streaming platforms, such as Youku, and iQiyi. Upon infecting Android devices, these malware fetch malicious payloads from http://***.itracker.cn:***, decode the binary, and execute the malicious script. Consequently, malware operators can publish codes on the C&C backends and execute arbitrary malicious vectors on infected devices.

Isolating Remote Payload Deployment Code. As the first challenge, to remediate this malware by reusing its remote payload deployment capabilities, incident responders must reverse engineer the malware sample and isolate the code implementing the payload deployment routine. This requires incident responders to decompile the binary and locate the system APIs that handle fetching, loading, and executing the remote payload. This task involves more than merely searching for API calls, as these events use generic system APIs that are also employed for other malware features. For instance, network request APIs can be used for fetching payloads or loading images. Incident responders must filter false positives by globally tracking data dependencies, pinpointing control flow logic, and combining execution traces with sandbox context information to identify code for payload deployment routines. As a result, the incident responder may find that the fake Youku malware uses a complex scheme: It pulls the payload from the C&C server and unzips it. After that, the script is decoded from a JSON text file and executed. In this case, incident responders cannot miss *even a single* event along the remote payload deployment routine. Even worse, code obfuscation techniques deployed by the malware operators make this task even more challenging.

```
template = function(){
    // Variable to be updated by incident responders
    var urlToNotifyUser = "<notification_to_users>";
    var shouldNotifyUser = true;
    var shouldUninstall = true;
    // ECHO-generated payload deployment routine info
    var packageName = "tv.huohua.android.ocher";
    // context-switching interface info
    var jsiObjectName = "RequestInfoControllerBridge";
    var jsiExecApiName = "runCmd";
    // ECHO-generated code execution template
    // execute Linux shell command via context-swtiching interface
    if(shouldNotifyUser){ //execute code to notify end user
    var cmdNotifyUser = "am start -a android.intent.action.VIEW
    -d " + urlToNotifyUser;
    eval(jsiObjectName + '.' + jsiExecApiName)(cmdNotifyUser)
    }
    if(shouldUninstall){ //execute code to uninstall app
    var cmdDelete = "am start -a android.intent.action.DELETE -d
    package:" + packageName;
    eval(jsiObjectName + '.' + jsiExecApiName)(cmdDelete)
    }
    }();
```

Fig. 1: Remediation Payload For Fake Youku Malware.

Remediation Capability Profiling. Simply replacing the payload with arbitrary scripts is ineffective for remediation. Incident responders must determine the correct code to implement to invoke remediation capabilities. In general, the payload can only invoke particular system APIs that are available within the malware's execution context. We define the term *in-vivo influence* to describe these system APIs that the payload can reach within the malware's execution context. In the fake Youku malware, the payload is written in HTML and JS and is executed within a WebView. The malware implemented an interface function, called `runCmd`, which passes its parameter to the system API `Runtime.exec`. Because this interface can be invoked from the remote payload, the malware operators can run arbitrary Linux shell commands on the victim's devices. Ideally, a *remediation payload* would invoke this interface and execute Linux shell commands to collect user consent and automate malware removal. Unfortunately, to achieve this, incident responders must reverse engineer the malware to 1) derive the possible in-vivo influence to interact with the malware context, 2) analyze the remediation capabilities the payload can leverage, and 3) draft remediation code accordingly.

Remediation Payload Generation. Finally, with the drafted code for the remediation payload, incident responders must generate a binary that can be hosted on the C&C server. This involves traversing the malware's payload deployment routine and reversing the procedures. Without a predefined specification, this task requires manual effort from incident responders to tailor the payload for each case, potentially slowing the remediation process. Subsequently, incident responders may collaborate with ISPs or DNS providers to redirect the payload fetching traffic.

A. *Incident Response With ECHO*

To accelerate taking down malware, we propose ECHO. ECHO provides an automatic pipeline for frontend malware remediation. ECHO takes only the malware binary as input.

Delving into ECHO's internal operation, we developed a formal model for payload deployment routines within the malware, as detailed in §III-A. ECHO automatically derives this formal model from the Youku malware using program

[1] sha-256 hash: 5135210444ad90b3a0d5aa5bd64fb06fedae8b44d 0b35a6f7e14be6128b476cf, package name: *com.youku.phone*

analysis techniques, as detailed in §III-B. The formal model abstracts each remote payload deployment event, including binary downloading, unzipping, script file decoding, and script execution. For each event, ECHO's formal model captures essential information to enable remediation, such as the C&C server to be seized for payload deployment. Next, ECHO automatically detects the remote payload deployed in a context-isolated sandbox (i.e., JS code executed in WebView). ECHO identifies the in-vivo influence of the payload by pinpointing the context-switching interface and the remediation-capable APIs (see §III-C). ECHO saves incident responders the effort of manually reverse engineering the malware to collect this prerequisite knowledge for generating the remediation payload.

With both the model representing the malware's remote payload deployment routine and the payload's in-vivo influence, ECHO generates a remediation payload template. This is detailed in §III-D. For the Youku malware, ECHO generated the template shown in Figure 1. The remediation payload template reuses the `runCmd` interface to call remediation-capable APIs to execute Linux shell commands, as shown in Lines 9-10. Additionally, ECHO provides the commands to be executed in Lines 14-15 and 19-20. The payload template enables incident responders to choose the remediation capability to be used and to customize the code if needed. For example, Line 3 of Figure 1 allows for adding a URL to notify users about the infection. Based on the formal model, ECHO provides specific steps to package and host the payload on a specific C&C server URL. After using ECHO, incident responders can customize this payload template, test it before deployment, and redirect the C&C traffic to conduct the remediation. We demonstrate this process for Youku (notifying the user, collecting consent, and malware removal) in a demo video at https://www.youtube.com/@CyFILab.

B. Threat Model

Frontend Malware. ECHO aims to remediate malware that fetches and executes payloads from remote C&C servers. Advanced malware operators periodically update their malware and C&C servers, making it hard for incident responders to block the C&C infrastructure entirely. The delay between updates depends on each malware program implementation. Therefore, ECHO has per-device variable latency, which we further discuss in §VI.

Victim Device. We assume the victims are devices infected by frontend malware. Incident responders cannot access the globally distributed victims physically. Victims routinely install security mechanisms (e.g., antivirus) but still become infected. As such, incident responders cannot rely on end-host security mechanisms.

Incident Responder. This paper uses the term "incident responders" to represent legally authorized entities that can coordinate with stakeholders, such as internet service providers (ISPs), for malware remediation. For example, Microsoft and Google both have received legal permission to deploy targeted code on victim devices to remove botnets [21], [22]. As another example, during the Retadup takedown [23], the French police hired Avast to painstakingly manually perform a remote frontend malware removal, similar to ECHO's motivation. ECHO is designed to support such cases. §VII discusses the legal and ethical cases where ECHO is necessary. We assume that incident responders can capture a malware sample from a reliable source, e.g., antivirus software, user reports, or trustworthy vendors.

III. ECHO's Methodology

A. Deployment Routine Formal Modeling

This research proposes a generic approach to malware remediation: reusing the remote payload deployment routine via remediation payload generation. Achieving a generic solution, however, remains an open research challenge because of the arbitrary freedom malware authors have in implementing remote payload deployment routines. Despite the diversity in code implementations, malware must conform to lifecycle stages to fetch, load, and execute remote code. Moreover, these lifecycle stages must exhibit specific data dependencies and control flow within and between each stage.

To this end, to accomplish ECHO's objective, we propose a graph-based formal model to abstract the malware's remote payload deployment routine. §III-B will find every remote payload routine implemented by a malware sample and instantiate ECHO's formal model for each, using a combination of binary program analysis techniques. Notably, ECHO's formal model identifies the payload's entry point method, which transfers execution from the malware's code to the payload. §III-C will utilize the formal model's entry points to find the reachable remediation capabilities. §III-D will leverage ECHO's formal model to generate the remediation payload template, facilitating the customization, packaging, and deployment of the remediation payload.

Overall, ECHO models malware's payload deployment routine with a directed acyclic graph, denoted as G. Each payload deployment-related event (represented by API calls and code sequences) is modeled as an annotated vertex ($v \in V$). Edges, denoted by $e \in E$, represent the execution path between two vertices with data dependency information. ECHO further uses *Assertions* for each edge to rule the conditions to be met by the remote payload when being deployed by the routine. The following details the vertex and edge definitions and how they construct the model's graph.

1) Annotated Vertices: ECHO models a vertex from a payload deployment-related event, which is either a system API call (e.g., network request sending call) or recursive code sequences (e.g., recursively decoding payload bytes in a fixed-length buffer). These concrete APIs or code sequences are denoted as $v.api$. Additionally, ECHO models each vertex with context information correlated to the API calls, called vertex annotations. These annotations are dynamically collected from the runtime sandbox context and represent vertices' runtime state. With payload fetching, loading, and execution as the three essential lifecycle stages for payload deployment routines, Table I lists ECHO-defined vertex types for each stage. Columns 1 and 2 show the vertex type and the symbol. Columns 3 and 4 list the vertex annotations and their symbols. ECHO also defines edge-in and edge-out assertions corresponding to each vertex type in Columns 5 and 6, respectively. We discuss them further in §III-A2.

TABLE I: Definition Of Payload Deployment Routine Vertices, Vertex Annotations, And Edge Assertions.

Vertex Type	Vertex Symbol	Vextex Annotations	Annotation Symbols	Edge-In Assertion^{1,2}	Edge-Out Assertion^{1,3}
Payload Fetching Stage					
Network Request Sending	v^f_{req}	backend URL communication protocol HTTP Session	url p s	N/A (Root Node)	$v.url \neq \phi \ \wedge v.s = v_{snk}.s \wedge$ $typeof(v_{snk}) = v^f_{res}$
Request Response Handling	v^f_{res}	response headers response content/binary HTTP Session	h b s	$typeof(v_{src}) = v^f_{req} \wedge$ $v.s = v_{src}.s$	$v.h.state = success \wedge$ $v.b \neq \phi \wedge$ $v.b = v_{snk}.b$
Payload Loading Stage					
Write Binary to File	v^l_{fw}	file path file binary	fp b	$v.fp \neq \phi \wedge v.b = v_{src}.b$	$fileExist(v.fp) \wedge$ $v_{snk}.fp = v.fp$
Read Binary From File	v^l_{fr}	file path file binary	fp b	$fileExist(v.fp) \wedge$ $v.fp = v_{snk}.fp$	$b \neq \phi \ \wedge \ v.b = v_{snk}.b$
Binary Decoding	v^l_{dec}	decoding algorithm decoding key pre-decoding binary post-decoding binary	alg k b_{pre} b_{pst}	$v.b_{pre} = v_{src}.b \wedge$ $(\neg v.alg.needsKey \ \vee$ $v.k \neq \phi)$	$v.b_{pst} \neq \phi \wedge$ $b_{pst} = v_{snk}.b$
Binary Segmentation	v^l_{seg}	segmentation index pre-decoding binary post-decoding binary	idx b_{pre} b_{pst}	$v.b_{pre} = v_{src}.b \wedge$ $v.idx \neq \phi$	$v.b_{pst} = v_{snk}.b$
Integrity Verification	v^l_{verify}	algorithm key or hash binary verification result	alg k b res	$v.alg \neq \phi \wedge$ $v.b = v_{src}.b \ \wedge \ k \neq \phi$	$v.res = true$
Payload Execution Stage					
Script Code Execution	v^e_{sce}	script binary entry point method context-crossing interfaces	b epm i	$v.b = v_{src}.b \wedge$ $scriptExecutable(v.b) \wedge$ $methodDefined(v.epm)$	$methodCalled(v.epm)$
Binary Code Loading	v^e_{bcl}	binary compiled class	b cls	$v.b = v_{src}.b \wedge$ $binaryCompilable(v.b)$	$typeof(v_{snk}) = v^e_{exe} \wedge$ $cls \neq \phi \ \wedge v.cls = v_{snk}.cls$
Entry Point Method Execution	v^e_{exe}	compiled class entry point method	cls epm	$v.cls = v_{src}.cls \wedge$ $methodDefined(v.epm)$	$methodCalled(v.epm)$

1: We use v in edge-in and edge-out assertions to represent this vertex to distinguish it from the other vertex in a potential edge.
2: We use v_{src} in edge-in assertions to represent the source vertex of a potential edge going into this vertex.
3: We use v_{snk} in edge-out assertions to represent the sink vertex of a potential edge out of this vertex.

In the payload fetching stage, ECHO models events related to malware sending network requests (v^f_{req}) and handling responses (v^f_{res}), as shown in Rows 1 and 2 in Table I. Alongside these vertices, ECHO collects information about the C&C hosts and the raw payload from the response. For the payload loading stage, ECHO models all payload-manipulating vertices. These vertices encompass the fundamental events in which the malware converts the raw responded payload into executable binary code. Specifically, such events include file I/O (v^l_{fr} and v^l_{fw}), decoding (v^l_{decode}), segmenting (v^l_{seg}), and integrity verifying (v^l_{verify}) as shown in Rows 3-7 in Table I. For the payload execution stage, ECHO models payload execution events for code written in both script languages (e.g., JS, Python) and compiled languages (e.g., C++, Java). For script-based payloads, ECHO models the script code execution APIs (v^e_{sce}). Additionally, for precompiled payloads, ECHO models the system APIs responsible for interpreting the payload into executable code (v^e_{bcl}) and the event of executing it (v^e_{exe}). In both cases, ECHO identifies the API implemented in the remote payload that is called from the malware's original code when the payload is executed. These APIs are referred to as *entry point methods*. ECHO leverages entry point methods to identify the in-vivo influence for the remediation payload, as detailed in §III-C.

2) *Data Dependency Edge With Assertions:* ECHO uses a directed edge to model the data dependency between two vertices. An edge is defined as $e = \{v_{src}, v_{snk}, d\}$, where v_{src} and v_{snk} represent the source and sink vertices and the data dependency context between two vertices (d). The data dependency represented by an edge can be a shared binary, pointer, or file path referring to the same data used by two vertices. As an example, for an edge representing the malware firstly handling a network response (v^f_{res}) and then writing the file to local storage (v^l_{fw}), the binary shared by both vertices is considered the data dependency context d on the edge.

However, the existence of a data dependency d between two vertices does not guarantee that this path can be satisfied during remediation payload deployment. To solve this challenge, ECHO's model represents predefined conditions as edge-in assertions for the sink vertex and edge-out assertions for the source vertex. Table I shows these assertions for each vertex type in Columns 5 and 6.

As an example, malware fetches a payload from one C&C server and a data file from another C&C server (two v^f_{res} vertices), and stores both buffers in a string array. When the payload content is executed (v^e_{sce}), data dependency analysis will lead back to both C&C servers. During dynamic analysis (§III-B), ECHO leverages the assertions to verify which server is used to deploy the remote payload. By checking the assertions, ECHO will find that the edge-in assertion for v^e_{sce} that $v.b \ \neq \ v_{src}.b$ (Row 8 in Table I) will fail for the data dependency path to the wrong C&C server. In §III-B, we provide more details about how assertion verification contributes to model instantiation.

3) ECHO's Formal Model: As a result, a valid ECHO model ($G = (V, E)$) must fulfill two conditions: 1) The model contains at least one vertex to send a payload fetching request (v^f_{req}) and one vertex to execute the payload (v^e_{exe}), and 2) there is at least one path that consists of edges in E that connect the payload fetching and payload execution vertices and have all edge assertions met. As a result, ECHO's model merges all identified execution paths into a graph. Noticeably, the graph can exhibit more complexity when the payload deployment routine implemented by the malware involves additional payload loading procedures. For example, when the malware pulls payload signatures from a remote server with an additional network request and verifies the payload against it, the model will include the additional payload fetching and loading vertices with corresponding edges. We further discuss ECHO's solution to overcoming the technical challenges posed by deriving the graph-based model out of payload deployment routines in §III-B.

B. Formal Model Instantiation

To derive the payload deployment routine, ECHO requires only malware as the input. Specifically, we conclude this procedure in three steps. First, without prior knowledge from the input malware, ECHO must model the malware's code into vertices to represent lifecycle events. Noticeably, this presents a challenge beyond simply matching the called API with a vertex type defined in Table I. This requires ECHO to combine the API signatures and the context information collected from the runtime for vertex annotations. Secondly, ECHO must identify the data dependency between vertices for edge generation. Thirdly, ECHO must find complete execution paths representing the malware's payload deployment routines from fetching the remote payload to executing it. This involves generating paths in G from edges in E and validating them through assertion accumulation. Unfortunately, to our knowledge, no prior work can solely overcome challenges in these tasks. As a result, ECHO proposes a program analysis module to derive the formalized model.

Technically, ECHO's vertex modeling requires a sandbox with a code hooking technique to log context information from particular system API calls. ECHO predefines the rules to map the system API and the context information to derive various vertex types. Besides, ECHO enables a force execution technique to actively trigger payload deployment code. For steps 2 and 3, ECHO accomplishes this by first identifying the potential execution paths between the payload fetching request vertex (v^f_{req}) and payload execution vertex (v^e_{exe}), and then removing the false-positive paths by verifying the edges along the paths with the accumulated assertions. ECHO achieves this with a hybrid data dependency analysis and a customized assertion verifying mechanism. In the following, we present ECHO's approach as a platform-agnostic solution with the necessary techniques for each step. Next, we further show our prototype implemented on the Android platform with technical details in §III-B3.

1) Vertex Instantiation: In general, for each ECHO-hooked API, ECHO injects the tailored code before and after the execution of the API and collects the call stack trace. Tailored to each vertex type, ECHO follows the vertices' definitions in Table I and models them from the malware's code as follows:

Network Request Sending (v^f_{req}) - This vertex represents the malware-sending network requests to a backend. ECHO models this vertex by hooking request-sending APIs and logs the annotations of remote URL and request headers.

Network Response Handling (v^f_{res}) - This vertex represents the malware handling the response. ECHO hooks the APIs of getting the response body, headers, and results separately and merges the results of these API calls into a single vertex. To be noticed, for frameworks implementing an asynchronous response handler interface, ECHO needs to hook the actual customized handler method that implements the interface.

Writing Binary to File (v^l_{fw}) - ECHO hooks system APIs for writing the binary from a buffer to a local file to track file-writing behaviors. This vertex is usually identified when malware caches the downloaded binary locally.

Read Binary from File (v^l_{fr}) - ECHO hooks file reading APIs to identify malware loading a local file. Such a vertex can be identified when the malware dumps the binary locally and loads it asynchronously in the payload deployment routine.

Binary Decoding (v^l_{dec}) - This vertex covers a variety of malware operations to modify the content of a binary with either a decoding procedure (e.g., `base64`) or a decryption procedure with a key (e.g., encrypted unzip, AES decryption). For decoding cases done with a system API, ECHO can hook particular APIs to gather the algorithm ($v^l_{dec}.alg$), the encryption key ($v^l_{dec}.key$), and the binary before and after decoding. However, for malware implementing customized encoding, such as using XOR with a constant key, ECHO can collect the key during dynamic analysis.

Binary Segmentation (v^l_{seg}) - This vertex covers the behavior of malware segmenting a binary and collects part of it. For example, the malware may fetch a remote JSON object and dump an executable payload value with a 'dictionary' key. Also, this covers the case in which the malware gets a substring of a text payload. ECHO hooks particular APIs to gather segmentation indexes and the binaries before and after the segmentation.

Integrity Verification (v^l_{verify}) - Considering that the frontend may verify the integrity of the payload before execution, ECHO hooks the APIs for verification to collect the algorithms, the key, and the verification results.

Script Code Execution (v^e_{sce}) - ECHO hooks system APIs for loading and executing the payload in script languages. Meanwhile, ECHO tracks the entry point method enabled in the executed script. If the script has no entry point method defined but executes the code in sequence, ECHO represents the entry point as a dummy main method. ECHO also pinpoints the context-cross interfaces.

Binary Code Loading (v^e_{bcl}) - ECHO hooks system APIs for loading precompiled binary into the malware's runtime environment, this is only specific to the payload written in a compiled language (e.g. C++, Java).

Entry Point Method Invocation (v^e_{exe}) - ECHO hooks the entry point method calls of a compiled payload and gathers the signatures. Notably, ECHO leverages the call stack trace to resolve the method caller-callee relationship to determine whether a call is an entry point for the remote payload.

In the sandbox, ECHO logs the corresponding API call and logs the necessary information for annotations associated with each vertex type. Besides, ECHO keeps tracking the temporal sequence of each log. We represent the sequence as an execution trace, denoted by ET, with each logged vertex denoted by $v_{et} \in ET$. We further showcase our implemented sandbox prototype in §III-B3 and the Android-specific API-to-vertex mapping list in §A.

2) Graph Generation: ECHO's strategy of graph generation comes with three steps. ECHO first identifies the potential execution paths of the malware fetching and executing the payload. Second, ECHO removes false-positive paths by accumulating the assertions for each path. Third, ECHO merges the payload deployment routines' paths into a graph as the formal model.

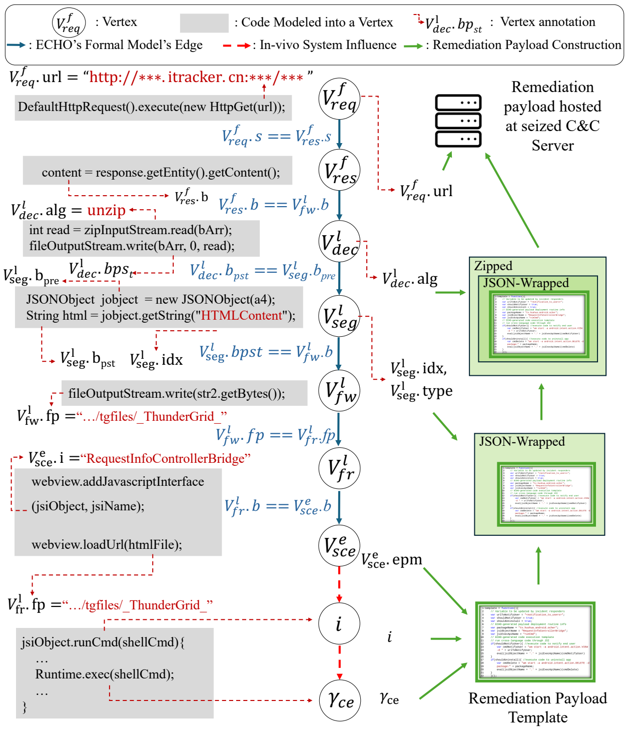
Initially, ECHO tracks the data flow with taint analysis. Specifically, ECHO takes v^f_{req} as source vertices and v^e_{exe} as sink vertices and finds all path-sensitive data flows in between. As a result, ECHO collects a list of data dependency paths (represented as ddp) for each pair of sink and source vertices, denoted by $DDP = \{ddp|ddp = (v^f_{req}, v^e_{exe}, V_{ddp}, E_{ddp})\}$. Here, V_{ddp} denotes a list of vertices statically resolved along with the data dependency paths, and E_{ddp} represents the edges on the path that connect two vertices from V_{ddp}. For each vertex $v_{ddp} \in V_{ddp}$ of a data dependency path, ECHO resolves the caller-callee relationship from the path (ddp) to derive a static call stack trace. Noticeably, this enables ECHO's generality even when only static taint analysis is available (as shown in §III-B3). However, the static call stack trace can be substituted when the dynamic taint analysis could capture the runtime call stack trace.

Next, ECHO derives the model from the list of data dependency paths (DDP) and the execution trace (ET) with the formal model generation algorithm shown in Algorithm 1. As the input, the algorithm assumes the data dependency is collected independently and does not contain vertices' annotation information. To start with, for each data dependency path $ddp \in DDP$, ECHO accumulates the assertion along the path to match the vertices from the execution trace ET to each statically resolved $v_{ddp} \in V_{ddp}$ (Lines 39-42). Specifically, this is done with a backtracking algorithm, as Lines 24-38 show in Algorithm 1. Toward each ddp, this function takes its statically resolved vertices list ($ddp.V_{ddp}$), a backtracking index (initially as 0), and a temporary data dependency path (d_{tmp}) to cache the processing results. Traversing each static vertex v_{ddp} in sequence, ECHO first matches all execution trace vertices that represent the same event as v_{ddp} with the function declared in Lines 2-10. The matching is accomplished by checking the stack traces and APIs from v_{ddp} against every v_{et} in ET (Line 6).

Next, ECHO applies the backtracking logic to test each matched v_{et} with assertion verification, as done in Lines 12-22 in Algorithm 1. This is done by verifying the edge-in and edge-out assertions (predefined in Table I) for the edge between the last vertex in d_{tmp} and the tested v_{et}

Algorithm 1: Formal Model Generation Algorithm

```
   Input: $ET$: Dynamically captured execution trace
   Input: $DDP$: List of data dependency paths from $v^f_{res}$ to $v^l_{int}$
          pairs
   Output: $G$: Formal Model Instance
   //Declare an empty set of valid data dependency paths
 1 $D \longleftarrow \varnothing$;
   //Function for filtering vertices from dynamic
     execution trace, which has the same event as $v_{ddp}$
 2 Function FindVerticesForEvent($v_{ddp}, ET$):
 3     $V_{et} \longleftarrow \varnothing$;
 4     for $v_{et} \in ET$ do
           //Check called API and stack trace
 5         if $v_{et}.api == v_{ddp}.api$ &&
 6           $v_{et}.callStackTrace == v_{ddp}.callStackTrace$ then
 7             $V_{et}.add(v_{et})$;
 8         end
 9     end
10     return $V_{et}$;
11 end
   //Verify assertions between two vertices from
     execution trace with their annotations' information
12 Function VerifyEdgeAssertions($v_{src}, v_{snk}$):
       //Dummy source vertex is always valid.
13     if $v_{src} == dummyHead$ then
14         Return $true$;
15     end
16     $assertionSet \longleftarrow union(v_{src}.edgeOutAssertions,$
           $v_{snk}.edgeInAssertions)$;
17     for $a \in assertionSet$ do
18         if !$isAssertionMet(a, v_{src}, v_{snk})$ then
19             Return $False$;
20         end
21     end
22     Return $True$;
23 end
   //Backtracking function for matching execution traces
     vertices to $ddp$ and verifying assertions.
24 Function $MatchVerticesOnPath(V_{ddp}, index, d_{tmp})$:
25     if $index == V_{ddp}.length - 1$ then
26         $D.add(d_{tmp}.deepcopy())$;
27         Return;
28     end
29     $v_{ddp} \longleftarrow V_{ddp}.getItemAt(index)$;
30     $V_{et} \longleftarrow FindVerticesForEvent(v_{ddp})$;
       //Verify the assertions between each $v_{et}$ and the
         last vertex in $d_{tmp}$.
31     for $v_{et} \in V_{et}$ do
32         if VerifyEdgeAssertions($d_{tmp}.lastVertex, v_{et}$) then
33             $d_{tmp}.appendVertex(v_{et})$;
34             $MatchVerticesOnPath(V_{ddp}, index + 1, d_{tmp})$;
35             $d_{tmp}.removeLast()$;
36         end
37     end
38 end
   //For each $ddp \in DDP$, check if all path events have
     matched vertices $v \in V$ with all assertions met.
39 for $ddp \in DDP$ do
       //The $d_{tmp}$ tracks temporary data dependency path
         with assertion met in backtracking algorithm.
40     $d_{tmp} \longleftarrow newListWithDummyHead$;
41     $MatchVerticesOnPath(ddp.V_{ddp}, 0, d_{tmp})$;
42 end
   //Merge all data dependency paths in $D$ into $G$
43 $G \longleftarrow \phi$;
44 for $ddp \in D, e \in ddp.E_{ddp}$ do
45     $G.addVertex(e.v_{src})$;
46     $G.addVertex(e.v_{snk})$;
       //Add edge to graph with source and sink vertices
         and data dependency context $d$ on the edge
47     $Graph.addEdge(e.v_{src}, e.v_{snk}, e.d)$;
48 end
49 Return $G$;
```



```
DefaultHttpRequest().execute(new HttpGet(url));

content = response.getEntity().getContent();

int read = zipInputStream.read(bArr);
fileOutputStream.write(bArr, 0, read);

JSONObject  jobject  = new JSONObject(a4);
String html = jobject.getString("HTMLContent");

fileOutputStream.write(str2.getBytes());

webview.addJavascriptInterface
(jsiObject, jsiName);

webview.loadUrl(htmlFile);

jsiObject.runCmd(shellCmd){
   …
   Runtime.exec(shellCmd);
   …
}
```

V^f_{req} : Vertex
: Code Modeled into a Vertex
$V^l_{dec}.bp_{st}$: Vertex annotation
: ECHO's Formal Model's Edge
: In-vivo System Influence
: Remediation Payload Construction
V^f_{req}. url = "http://***. itracker. cn:***/*** "
$V^f_{req}.s == V^f_{res}.s$
V^f_{res}. b
V^l_{dec}. alg = unzip
$V^f_{res}.b == V^l_{fw}.b$
V^l_{seg}. b$_{pre}$
$V^l_{dec}.bps_t$
$V^l_{dec}.b_{pst} == V^l_{seg}.b_{pre}$
V^l_{seg}. b$_{pst}$
V^l_{seg}. idx
$V^l_{seg}.bpst == V^l_{fw}.b$
V^l_{fw}. fp ="…/tgfiles/_ThunderGrid_"
$V^l_{fw}.fp == V^l_{fr}.fp$
V^e_{sce}. i ="RequestInfoControllerBridge"
$V^l_{fr}.b == V^e_{sce}.b$
V^l_{fr}. fp ="…/tgfiles/_ThunderGrid_"
V^f_{req}
V^f_{res}
V^l_{dec}
V^l_{seg}
V^l_{fw}
V^l_{fr}
V^e_{sce}
i
γ_{ce}
Remediation payload hosted at seized C&C Server
V^f_{req}. url
V^l_{dec}. alg
V^l_{seg}. idx,
V^l_{seg}. type
V^e_{sce}. epm
i
γ_{ce}
Zipped
JSON-Wrapped
JSON-Wrapped
Remediation Payload Template

Fig. 2: Formal Model Instantiated From The Youku Sample. The linked list in the middle is the formal model with identified in-vivo system influence (red dotted line), with the payload deployment routine shown on the left and the remediation payload generation procedure on the right.

(Line 32) with their annotation information. If the verification passes, ECHO appends the new v_{et} to the d_{tmp} (Line 33) and processes the next v_{ddp} in V_{ddp} (Line 34). If it reaches the last vertex in V_{ddp}, the d_{tmp} now stores a data dependency path with each $v_{ddp} \in V_{ddp}$ matching a $v_{et} \in V_{et}$ and meets all assertions in Line 24. ECHO saved the deep copy of d_{tmp} as a valid data dependency path, which represents a payload deployment routine implemented by the malware (Line 26). Finally, in Lines 43-49, ECHO merges all edges from all valid data dependency paths into the graph G as the formal model.

3) Android Running Example: In the following, we walk through ECHO's model deriving procedures on our running example with our prototype on the Android platform. Figure 2 shows the derived formal model (as the linked list shown in the middle), which represents the identified payload deployment routine at the left. The right part of Figure 2 represents ECHO's procedure to generate the remediation payload, as detailed in §III-D. ECHO first derives the payload fetching and execution vertices by hooking network request APIs (e.g., `DefaultHttpRequest.execute`) and response handler (i.e., `HttpEntity.getContent`) to get the v^f_{req} and v^f_{res} vertices as well as their annotations. With a similar approach, ECHO hooks the WebView's `loadUrl` API. As it loads and executes the script from a local file or a remote URL directly, ECHO models it into a combination of a file-reading vertex (v^l_{fr}) and a script-execution vertex (v^e_{sce}). Specific to Android, this uses a component-level force execution sandbox with the API hooking capability, for which we provide additional implementation details in §B. In addition, as a unique challenge in the Android platform, ECHO features a multi-threading stack trace logging algorithm (as detailed in §C) to capture the complete stack trace for each vertex logged along the execution trace.

To derive the data dependency paths between payload fetching vertices (v^f_{req}) and payload execution vertices (v^e_{exe} and v^e_{sce}) in Figure 2, ECHO deployed a hybrid data flow analysis, which uses the dynamic execution trace to improve static taint analysis results (§D). Next, by applying the Algorithm 1, ECHO removes the false-positive data dependency paths (ddp) and ends up with the formal model shown in Figure 2. Specifically, ECHO finds that, after fetching the payload, the malware unzips the raw payload (v^l_{dec}), extracts the `html` content from the unzipped JSON binary (v^l_{seg}), and dumps the file to the local ($v^l_f w$) before being loaded by WebView (v^l_{fr} *and* v^e_{sce}). This formal model enables ECHO to pinpoint the in-vivo influence of the remote payload and generate the remediation payload.

C. In-Vivo System Influence Analysis

As ECHO finds the payload deployment routine with entry point methods, its next step is to pinpoint the in-vivo influence of the remediation payload. Specifically, with the capability of reusing the payload deployment routine enabled by the malware, incident responders still cannot simply replace the malicious payload with a remediation payload, which can lead the takedown attempt to a dead end. Using the Youku malware as an example (§II), with a JSI payload deployment routine, it is not feasible to implement arbitrary Java code in an `html`-based payload for remediation. ECHO must identify how the remediation payload can affect the malware when the malware executes the payload by calling the entry point method; we call this the payload deployment routine's in-vivo influence.

Pinpointing the in-vivo influence requires identifying *remediation-capable APIs* (denoted by γ). They refer to system APIs that can be directly or indirectly called in the payload to interact with the malware's runtime context and induce malware remediation consequences (i.e., notifying end users about the infection and removing the malware from the victims' device). This requires that ECHO accomplish two technical checkpoints: 1) ECHO needs to check the context limitation of the payload and 2) identify the system API that can be utilized by the remote payload if it is in a limited context. For the first checkpoint, ECHO checks this by awarding the runtime context when the entry point method (v^e_{exe}) of the remote payload is called. If the context is the same as the malware, ECHO considers the payload to have the same privilege, and the payload can implement arbitrary code. However, with the payload having an isolated context, ECHO must track the predefined routines enabled by the malware to engage the context for remediation.

Context-Switching Interface. In general, if malware attempts to deploy a payload written in other languages to enable the payload to interact with the malware's context, it must predefine APIs that can be invoked from the payload's context to execute its method body in the malware's context. These APIs must be implemented in the malware's code prior to the remote payload deployment. As these APIs enable the payload to execute code crossing the context boundary, we call such APIs predefined in the malware's code as *Context-Switching Interfaces*, denoted by i. In this case, the only approach that enables the remediation payload to influence the malware is calling the context-switching interfaces in the payload and passing the parameters to the predefined system APIs in the interfaces' method body.

With a similar technique introduced in §III-B, ECHO achieves this by tracing the data dependency paths between the parameters of the context-switching interfaces (i) and the remediation-capable APIs (γ) and verifying the vertices along the path. With this, ECHO can also handle cases with parameter manipulation (e.g., string segmentation and binary decoding) by checking the payload loading vertices along the path and ensuring the parameter propagation is valid.

1) Remediation-capable API Finding: With our prototype, specific to Android WebView's isolated context, ECHO defines five types of remediation-capable APIs for JSI payload deployment routines. They are listed with the parameter-passing rules to be fulfilled below:

Command Execution (γ_{ce}) – With APIs executing Linux shell commands to impact the device directly, a command should be passed from i to γ_{ce} for remediation.

Application Termination (γ_{at}) - System APIs can be invoked to terminate the app, such as `System.exit`. In this case, no parameter is needed to shut down the malware.

Intent Control (γ_{int}) - This allows the app to open a new view or activity. For example, passing `Intent.ACTION_VIEW` and a URL as parameters, this API allows a device to show an alert page to users. Besides, an `Intent.ACTION_DELETE` intent can ask for users' permission to uninstall the malware. ECHO needs to ensure that the context-switching interface i can pass both the intent type and the optional parameter to the γ_{int}, such as the URL for the popup view.

Toast Message (γ_{tst}) - Toast APIs can be used to pop up a toast message passed from the entry point method on the screen. This enables ECHO to show a user an alert.

Load Webpage (γ_{web}) - WebView's `loadUrl` APIs can show users with webpages. This enables incident responders to collect users' content and provide malware-cleaning guidance. ECHO considers that a string representing the loaded URL must be passed from i.

As shown in Figure 2 for the running example, ECHO found that the malware passes the parameter from the method `runCmd`, which is implemented in the JSI context-crossing interface `RequestInfoControllerBridge` (i), to the remediation-capable API `Runtime.exec` (γ_{ce}). This enables incident responders to run *Linux* shell commands to remediate malware. Noticeably, this API list can be easily extended by adding more APIs, given ECHO's modular design.

2) Extending ECHO to Other Platforms: Although we showcase ECHO's in-vivo influence analysis tailored for Android with a JSI payload routine, a similar idea can be easily extended to other platforms and different payload deployment routines. For instance, with malware using an electron framework to build desktop malware, the payload written in web code is isolated and must engage the runtime context with the framework-enabled APIs. ECHO only needs to extend the prototype on the targeted platform and extend the remediation-capable API list. `shell.openExternal` in electron framework, for instance, has the same *Command Execution* capability as listed in §III-C1.

D. Remediation Payload Construction

With the payload deployment routine's formal model and the in-vivo influence analysis results, ECHO's last step is to generate a remediation payload and provide guidance for incident responders to customize, package, and test the remediation before deployment.

1) Remediation Payload Template: ECHO generates the therapeutic payload template in two steps: Based on the formal model and corresponding in-vivo influence results, ECHO tailors the remediation payload template toward each malware. With a payload deployment routine that executes a payload in a scripting language, ECHO generates the template to be executed by script execution vertices (v^e_{sce}). If the payload deployment routine takes a payload in a compiled language, ECHO generates the source code as the template, which can be compiled into a binary to be loaded and executed by v^e_{bcl} and v^e_{exe} vertices in the model. If the payload earns unlimited in-vivo influence, such as the payload being deployed with an RDCL routine, ECHO only needs to implement and export the entry point method. The entry point method's information can be retrieved from the entry point method execution vertex v^e_{exe}. Within the entry point method body, ECHO fills it with an ad-hoc template and enables incident responders to make further customization.

If the payload has a limited in-vivo influence and requires calling remediation-capable APIs (γ) by calling context-crossing interfaces (i), ECHO must generate the remediation payload template by passing essential parameters from i to γ. In this case, ECHO enumerates the remediation-capable APIs (γ) that can be indirectly invoked via calling the interfaces and implementing the parameter format by reversing the data dependency paths identified in §III-C. As the example in Figure 1, ECHO identifies the *Command Execute* capability, which enables the Linux shell command execution in the malware's context. In the template, Lines 8 and 9 implement the interface signature. Lines 13-14 and 18-19 are tailored to pass the Linux command to the γ. This enables incident responders to both notify end users with a customized web page and to perform auto-deletion.

2) Remediation Payload Customization: Specific to our prototype, with an RDCL remediation payload that runs arbitrary code, incident responders can update the ad-hoc payload body accordingly to collect the user's consent and automate the frontend removal. With a JSI remediation payload, for different capabilities, as shown in Figure 1, ECHO still enables incident responders to add a customized user-notification URL. Lines 4-5 also allow incident responders to freely select the features they want to enable based on the payload's capability. As a result, ECHO provides incident responders with both capability and flexibility in a scalable manner. Next, ECHO backtracks the formal model to identify paths between payload fetching vertices (v^f_{req}) and the payload execution vertices (v^e_{exe}) and reversing the payload

TABLE II: Overall Takedown Routine Identification Results.

Family	**#Samples**	**RDCL Routines**			**JSI Routines**				**FSD**^{2}	**LSD**^{2}	**Takedown**^{3}**(%)**
		#R$^1_{unique}$	**#S**^{1}	**#B**^{1}	**Capabilities**^{4}	**#R**$^1_{unique}$	**#S**^{1}	**#B**^{1}			
hiddenapp	113	2	109	54	-	0	0	0	2022-08-30	2023-01-25	109 (96.46%)
shedun	94	9	67	6	-	0	0	0	2022-09-06	2022-09-22	67 (71.28%)
hiddad	89	1	57	25	-	0	0	0	2022-08-12	2023-01-25	57 (64.04%)
fakeadblocker	69	2	68	39	-	0	0	0	2022-09-01	2023-01-26	68 (98.55%)
skymobi	66	9	56	4	-	0	0	0	2022-09-06	2023-01-03	56 (84.85%)
grayware	48	3	30	11	Tst,Int	2	19	2	2022-10-09	2023-01-24	32 (66.67%)
spyagent	46	0	0	0	Tst,Int	2	31	1	2022-08-30	2023-01-26	31 (67.39%)
hiddenads	37	2	37	29	-	0	0	0	2022-08-30	2023-01-26	37 (100.00%)
smspay	35	3	12	2	-	0	0	0	2022-09-06	2023-01-24	12 (34.29%)
remotecode	29	1	29	20	-	0	0	0	2022-08-30	2023-01-26	29 (100.00%)
smssend	16	0	0	0	Int	2	2	2	2022-09-07	2023-01-22	2 (12.50%)
resharer	14	0	0	0	AT,Tst,Web	3	7	2	2022-08-14	2023-01-13	7 (50.00%)
metasploit	11	0	0	0	Int	2	2	4	2022-09-02	2023-01-16	2 (18.18%)
youku	7	0	0	0	CE	1	5	1	2022-09-03	2022-10-06	5 (71.43%)
robtes	7	0	0	0	Int	1	1	1	2022-10-25	2022-12-27	1 (14.29%)
masplot	5	0	0	0	Int	1	1	1	2022-09-08	2023-01-08	1 (20.00%)
smsbot	5	0	0	0	Int	1	1	1	2022-09-09	2022-11-05	1 (20.00%)
ramnit	4	0	0	0	Int	1	1	1	2022-09-14	2023-01-21	1 (25.00%)
nimda	3	0	0	0	AT,Tst	2	2	3	2022-09-01	2022-11-13	2 (66.67%)
backdoor	2	0	0	0	Tst	1	1	1	2022-09-08	2022-09-30	1 (50.00%)
hypay	1	0	0	0	AT	2	1	1	2022-09-27	2022-09-27	1 (100.00%)
silentinstaller	1	0	0	0	Int	1	1	1	2022-10-05	2022-10-05	1 (100.00%)
Total	702	18^5	465	136^5	**5**	23	75	22	2022-08-12	2023-01-26	523 (74.50%)

1: Numbers of unique routine (#R$_{unique}$), malware samples have RDCL (or JSI) routines (#S), and backends (#B).
2: First Seen Date (FSD) and Last Seen Date (LSD).
3: Number and percentage of samples can be taken down through identified routines.
4: Remediation-capable APIs, include WebView (Web), Intent (Int), Toast Message (Tst), App Termination (AT), and Command Execution (CE).
5: Several routines and backends are shared by multiple families. These are the sum of unique routines and backends.

loading sequences. As shown in the right side of Figure 2, the identified payload loading vertices from the model guides incident responders to package and deploy the remediation payload via template generating, encoding, zipping, and hosting it on the seized C&C servers, which is revealed by the annotation of the payload fetching vertices (v^f_{req}).

IV. Evaluation

Implementation. ECHO's prototype is implemented on the Android platform with multiple integrated modules. Our dynamic execution framework is complemented by Xposed [24] plugins (15k LOC). The framework's network uses a MiTM Proxy [25]. The guided taint analysis module is built on top of FlowDroid [26] with additional customized code (2,000 LOC). These modules are integrated into a modular pipeline scripted in Python to guide the analysis. We discuss extending ECHO to other scenarios in §VI-B. For hardware, we deployed our static analysis on an in-house cluster with five VMs. Each VM runs Ubuntu 20.04 with 120GB memory and 16 vCPUs. Our dynamic analysis framework is deployed on our testbed, which features 10 customized Pixel 3 phones and a local desktop that runs Ubuntu 20.04 with 16GB memory and quad-core 2.4 GHz Intel Xeon CPUs. The processing time per sample varied by its code complexity, ranging from 2 to 10 minutes for dynamic analysis and 3 to 60 minutes for static analysis.

Dataset. To evaluate the prevalence of Android remote code reflection malware, we deployed ECHO against 702 malware samples. Our team and Netskope pulled the Android malware sample stream from *VirusTotal* [27] daily from Aug. 10, 2022, to Jan. 26, 2023, resulting in over 20,000 random malware samples in our initial dataset. We filtered each sample based on *VirusTotal*'s report with over five detection engines reporting it as malicious [28]. To find samples that may execute remote payloads, we compiled a payload deployment-related API list as shown in §A, performed a simple dynamic sandboxing execution, and selected samples that invoked one of those APIs. This yielded 1,057 samples. We reverse-engineered the samples manually and found that the sandbox produced 355 false positives (i.e., the malware executed an API from the list but not related to payload reflection). Our results were confirmed with industry and *VirusTotal* reports, and this became our ground truth. We labeled the malware families with *VirusTotal* reports and AVClass2 [29]. This resulted in a dataset of 702 samples across 22 families, including 580 RDCL candidates and 170 JSI candidates, with 48 overlapping.

A. Takedown Routine Identification

To evaluate the performance, ECHO was deployed on the 702 candidates from the evaluation dataset to identify RDCL and JSI routines, as detailed in Table II. The first two columns show the malware family names and the sample counts in the dataset. Columns 3-5 and 7-9 show the number of identified RDCL and JSI routines (#R$_{unique}$), malware samples (#S), and identified payload hosting backends (#B). Column 6 shows the accessible remediation capabilities for the JSI routines. Columns 10-11 present the First and Last Seen Date (FSD and LSD) from *VirusTotal* for each family. The last column lists the count and percentage of samples that can be taken down.

In the Total row of Table II, ECHO can successfully take down 523 of 702 samples (74.5%). From the 523 total, ECHO

TABLE III: Top 15 Payload Hosting Backends.

Backend	IP	#P[1]	#S[1]	#F[1]	Routines		TLS[2]	Geo[2]	Ownership	FSD[2]	LSD[2]	Takedown[3]
					RDCL	JSI						
**iaocft.com	*.*.229.90	1	77	2	3	0	False	HK	DXTL-HK	2022-09-06	2022-09-19	77
**shui.com	*.*.7.123	1	47	3	5	0	False	US	HK Megaplayer	2022-09-06	2023-01-03	47
**qq.com	*.*.226.35	1	17	1	0	1	True	CN	ChinaNet	2022-09-09	2022-09-30	17
**ch.cn	*.*.216.185	1	17	1	2	0	False	CN	Huawei Cloud	2022-10-09	2022-10-09	17
**xapt.com	*.*.125.182	4	13	1	1	0	True	NL	Hostinger	2022-11-10	2022-12-01	13
**x5.com	*.*.58.41[4]	4	11	2	4	0	False	CN	CloudVSP	2022-09-06	2022-09-18	11
**ks.cn	*.*.249.217	1	6	1	0	2	False	CN	China Telecom	2022-08-14	2022-10-19	6
**sullion.pro	*.*.36.203	1	7	2	1	0	True	US	Cloudflare	2022-10-02	2023-01-26	7
**ione.club	*.*.48.13	1	6	2	1	0	True	US	Cloudflare	2022-08-30	2022-11-10	6
**ningba.info	*.*.24.228	1	6	2	1	0	True	US	Cloudflare	2022-08-30	2023-01-01	6
**ianeeu.info	*.*.4.129	1	6	2	1	0	True	US	Cloudflare	2022-09-01	2023-01-18	6
**ckets.pro	*.*.59.132	1	6	2	1	0	True	US	Cloudflare	2022-09-04	2023-01-24	6
**ceme.info	*.*.58.122	1	6	2	1	0	True	US	Cloudflare	2022-09-04	2022-12-20	6
**esme.info	*.*.58.164	1	6	2	1	0	True	US	Cloudflare	2022-09-10	2022-11-13	6
**ker.cn	*.*.33.27[4]	1	5	1	0	1	False	CN	China Telecom	2022-05-01	2022-10-06	5

1: #P, #S, #F are shorts for numbers of payloads, malware samples, and families respectively.
2: Using TLS protocol (TLS), Geographic location (Geo), First Seen Date (FSD), and Last Seen Date (LSD)
3: The number of samples can be taken down by seizing the backend and deploy a remediation payload.
4: The backend has been taken down at the time of our analysis, and the IP information is based on DNS history.

identified 470 malware samples that can be removed and 53 that can only inform the user (e.g., provide instructions to remove manually). We manually confirmed these results. Specifically, 465 out of 580 (80.17%) RDCL candidate samples and 75 out of 170 (44.12%) JSI candidates can be taken down through ECHO's methodology. For the remaining 179 samples, ECHO reported no deployment routine that could be used for remediation. This included 115 RDCL samples using local resources and 95 JSI samples either loading local HTML files or lacking sufficient JSI capabilities.[2] As ECHO performed correctly on these samples, incident responders can still use the identified C&C backends for alternative takedown approaches.

For RDCL routines, ECHO found that malware samples in the same family share routine implementations. For *hiddenapp*, out of 113 samples, 109 samples share only two unique RDCL routine implementations, which connect to 54 C&C backends. Interestingly, the shared routine implementations can still connect to different C&C backends. For example, ECHO identified 29 *remotecode* samples sharing one routine implementation to fetch malicious payloads from 20 backends. In Table II, 18 RDCL routines are identified from 465 samples (in 9 malware families), exposing an opportunity for incident responders to reuse the payload generated by ECHO to perform the frontend remediation on a large scale.

Column 6 of Table II shows the JSI payload capabilities. ECHO found five malware families (Rows 3, 12, 14, 19, and 21) exhibiting App Termination (AT), WebView loading (Web), and Command Execution (CE). These capabilities reported by ECHO enable incident responders to disrupt and even remove the malware from infected devices. Additionally, ECHO found five malware families (22%) enabling APIs to show Toast messages and 10 families (45%) enabling Intent sending, which provides incident responders opportunities to notify the victims with either an alert message or a warning page. Notably, *grayware* malware possesses both RDCL and JSI routines, offering varied takedown strategies.

From the dataset, we also observed malware samples that are not part of well-known families. For these families, the malware samples may be implemented by different attackers and be more diverse. Such samples may or may not implement either anRDCL or a JSI routine. For example, the *smssend* family (Row 11) profits from silently sending SMS messages from infected devices [30]. ECHO detects two samples out of 16 implementing a JSI routine and thus applicable to ECHO's takedown solution. We manually verified that the rest of the samples are not capable of receiving and executing remote commands. We additionally validated ECHO to show that it could accurately identify deployment routines with low false positives and false negatives, shown in §E because of space constraints.

B. Payload Hosting

Table III presents the top 15 payload hosting backends, ranked by the number of samples ECHO found connecting to them. The results are aggregated by the effective second-level domain (ESLD). Columns 1-2 present the ESLD and resolved IP. We masked these to avoid their being used to target any still-infected frontend devices. For inactive backends, ECHO uses the full URL to indicate a unique payload and retrieves IP data from DNS history. Columns 3-7 details the number of hosted payloads, samples, families, and routines linked to each backend. Column 8 shows whether the malware used TLS. Geographic information (Geo) and Ownership are shown in Columns 9-10. Columns 11-12 show the FSD and LSD of the samples interacting with each backend. Column 13 enumerates samples that can be taken down via backend seizing.

In Row 2 of Table III, ECHO identified backend ***shui.com hosting one payload fetched by 47 malware samples across three families. Further analysis suggests that the payload fetching vector is performed by a shared SDK, hinting at a potential malware build kit. As shown in Rows 8-14, ECHO identified 43 *different* malware samples communicating with one Cloudflare subnet. Noting significant similarity in the C&C URL format, we reverse-engineered these payloads and found massive code sharing with minor

[2] 31 samples were candidates for both RDCL and JSI groups.

TABLE IV: Deployment Routine Implementation Results.

Type	Payload Deployment Routine	S[1]	F[1]	R[1]	TD[2]
RDCL	JSON → APK → Reflection	297	5	3	297
	APK → MD5[3] Verify → Intent	107	3	9	107
	APK → Reflection	30	3	2	30
	Zip → APK → Reflection	17	1	2	17
	DEX → Reflection	13	1	1	13
	Data→XOR[4]→DEX→Reflection	1	1	1	1
JSI	Zip →JSON →HTML→WebView	5	1	1	5
	HTML → WebView	70	19	22	70

1: Numbers of samples (S), families (F), and routines (R).
2: Number of samples that can be takendown.
3: MD5 algorithm is used for verifying the remote payload.
4: The data is decoded by XORing a constant pattern.

updates, possibly indicating a single malware campaign. Surprisingly, Row 3 highlights a subdomain of *qq.com*, one of the largest internet companies in China, which is abused by the 17 malware. While the URL is not deactivated, we are working with our industry collaborator to report this abuse. For geographic location distribution, the majority of C&C backends are located in the US (53%) and China (33%), emphasizing the challenges incident responders may face in blocking or sinkholing all C&C backends — highlighting the importance of frontend malware remediation.

C. Deployment Routine Measurement

To study payload deployment routines' implementations, we classified the identified routines from 523 samples into eight groups based on the fetched payload type, loading path, and execution method. Table IV lists the six RDCL (Row 1-6) and two JSI (Row 7-8) routine categories in Column 2. Columns 3-5 show the number of malware samples (S), families (F), and routines (R). The last column shows the number of samples with deployment routines that ECHO can remediate. Notably, ECHO found 297 out of 580 (51.21%) samples in five families that use JSON to wrap binaries, adhering to RESTful API practices [31]. In Rows 3-5, 60 samples with five routines fetch and execute zipped or plain-text remote binaries with code reflection. For JSI routines, ECHO found 70 malware samples directly loading remote HTML via WebView.

Payload Encoding. Our evaluation found only six malware samples from two families that encode payloads. Five samples from the Youku family (as introduced in §II) load HTML payloads with a complex encoding sequence (Zip → JSON → HTML → WebView). Another RDCL sample used XOR encoding with a constant pattern "UTF8." As shown in Table IV, ECHO reported these behaviors to incident responders with exact locations and data flows in the malware's code, helping incident responders to encode their payload before delivery (as we did in the Youku demo video in §II-A).

Payload Verification. ECHO found 107 malware samples from three families that use MD5 payload verification to check payload integrity. Remediating these malware samples is similar to Payload Encoding discussed above but also involves backend takedown efforts. First, incident responders can refer to ECHO's output to identify the C&C backend that delivers the verification signature (the "signature backend"). Next, incident responders can sign their remediation payload prior to delivery. Finally, during their existing backend takedown efforts, incident responders can seize the traffic from the signature backend and push the new verification signature for the remediation payload to the malware. The other 416 (79.54%) samples do not verify their payload, and no malware encrypted or signed the payloads, which aligns with the findings of prior work [16]. We discuss handling advanced attackers in §VI-A.

TABLE V: Malware Packing And Obfuscation.

Family	#Samples	#Obfus.	Packer	#Unpack
shedun	94	83	N/A	N/A
skymobi	66	66	N/A	N/A
resharer	14	14	MTP: 1	1
grayware	48	6	Jiagu: 6	6
smsreg	28	1	qdbh: 11	11
downloader	75	1	qdbh: 9	9
generickdz	11	0	qdbh: 8	8
spynote	9	3	AE[1]: 3	3
congur	2	2	AE[1]: 2	2
smssend	16	0	qdbh: 3	3
Total	363	176	43	43 (100%)

1:AE is short for *ApkEncryptor* packer.

TABLE VI: ECHO's FakeAdBlocker Analysis Results.

Package	`com.beacon.weather`
Backend	`****one.club`
RDCL Routines	**Routine**: JSON → APK → Reflection **Download**: `main.apk` **Class Name**: `b.b.a.v$d` **Method Name**: `checkServerTrusted`
Disruption Payload	1) Alert users 2) Uninstall frontend bots

D. Packing and Obfuscation

We evaluated ECHO's capability to manage packed and obfuscated malware samples, as shown in Table V. Each row shows the number of total samples and samples protected by packers or obfuscators in each family. Packers and obfuscators were labeled with APKiD [32], with results manually verified. Obfuscated samples have unpacked code but non-readable methods and fields. The last column shows sample counts that ECHO successfully unpacked.

Of all 702 malware samples, we found 176 obfuscated samples. Obfuscation poses no challenge to ECHO because its program analysis does not require easy-to-read code. ECHO also successfully handled all 43 packed malware samples from 8 families with four packers, successfully detecting these packers trying to load classes from encoded binaries.

V. Case Studies

A. New Backends and Routines Sharing

In this section, we showcase how ECHO could help incident responders remediate a *FakeAddBlocker* family that has 68 samples and talks to 39 different backends. Our evaluation highlighted a malware, called *com.beacon.weather*[3], which retrieves a payload from its C&C backend *****one.club and

[3] sha256 hash: 07e984b03d5a84dcfe8023adbae628a7b8089544b6a047b70beef40b1e2a869f

TABLE VII: Payload Hosting For Dexapt Botnet.

Dex Name	SHA256[1]	#Samples	ΔDays	Takedown
2021-09-22.dex	***589664	1	0	1
2021-09-23.dex	***589664	1	1	1
2021-09-25.dex	***589664	2	2	2
2021-10-04.dex	***589664	1	9	1
2021-11-03.dex	***114866	1	30	1
2021-11-17.dex	***9bad58	1	14	1
2021-12-09.dex	***9bad58	1	22	1
2021-12-24.dex	***6ed746	1	15	1
2021-12-27.dex	***6ed746	1	3	1
2022-01-10.dex	***6ed746	3	14	3
Total	4	13	12	13

1: Last 6 digits of SHA256.

employs RDCL routines for code execution, as detailed in Table VI. The backend communication is TLS-encrypted and the C&C backend dispatched a JSON response containing the embedded payload binary for payload fetching.

ECHO's *Payload Routine Identification* reported that the payload is decoded and stored locally as `main.apk`. The APK contains an API, `b.b.abv$d: void checkServerTruested`, which is called via reflection. Using this API, ECHO crafted an RDCL remediation payload that can trigger arbitrary behaviors (e.g., uninstall itself, alerting users) by reusing the malware's RDCL routine. We further queried *VirusTotal* for siblings of this sample and found an additional 102 samples. From these, ECHO identified another 15 backends hosting the same payload and confirmed that the same takedown method applies to all samples. This demonstrates ECHO's capability to swiftly neutralize malware programs even if they shift C&C servers and fetch updated payloads, enabling responders to consistently execute remediation upon malware detection.

B. Payload Updating Over Time

In our study, ECHO identified the domain ***xapt.com* as a C&C backend distributing frequently updated malicious payloads. Cisco Talos [33] also flagged this domain for its suspicious activities. As outlined in Table VII, 13 samples were found fetching four unique DEX payloads from this backend. Column 1 shows the hosted DEX filenames, which also reveals the updated timeline of the malware operators. Despite these files being actively available, Column 2 shows the last six digits of payload hashes. Column 3 lists the number of frontend samples from our dataset fetching each payload. Column 4 highlights the days between each payload update. Column 5 enumerates samples that ECHO can remediate.

The malware operator consistently updated the payloads over a period of 110 days, from September 22, 2021, to January 10, 2022. As shown in the Total row of Table VII, the malware operator updated the hosted payload around every 12 days. ECHO's output showed that these payloads utilize a consistent entry point accessed by the malware access via code reflection. Intuitively, this is the best design for malware operators: They can regularly update the payload and do not need to update the malware, which can continue to reflect the same entry point. With ECHO's insight, incident responders can leverage this to distribute a single remediation payload to all infected devices.

VI. Discussion

A. ECHO Against Advanced Attackers

TLS-Encrypted Traffic. As demonstrated in §IV-B, using application-level encrypted protocols, specifically HTTPS, is a prevalent method for encrypting traffic to prevent monitoring. While this does not affect ECHO's capability to discern payload deployment routines and generate remediation payload templates, incident responders need to undertake additional measures to deploy a remediation payload. To elaborate, when redirecting a payload request via HTTPS, incident responders must collaborate with a certificate authority (CA) to invalidate the existing certificate and establish a new one for the redirected backend. This has been previously employed in botnet sinkholing against TrickBot [34] and Glupteba [35].

Symmetric Encryption. To enhance both integrity and confidentiality, attackers could encrypt and sign the payload before deployment. By using symmetric encryption algorithms (e.g., AES), the payload can be encrypted by the C&C backend and decrypted by the malware with the same key. In this case, ECHO identifies the payload deployment routines and reports encryption usage. Similar to the XOR encoding case in §IV-C, incident responders can extract the encryption key by following the payload deployment routine and data flows reported by ECHO. Subsequently, incident responders can use the extracted key to encrypt the remediation payload prior to deployment.

Asymmetric Encryption. While not observed in our dataset, prior research [11] suggests that attackers could sign payloads using asymmetric encryption. This poses challenges for incident responders in deploying remediation payloads without the requisite signing key. These challenges are not specific to ECHO, and handling strong encryption in any part of an attack is known to require additional human effort. In such cases, ECHO will remain instrumental in helping incident responders swiftly generate remediation payloads and identify C&C backends. If the malware fetches the public key from a C&C backend, remediation is the same as in the MD5 verification case presented in §IV-C. Alternatively, malware could hard-code the public key, but this would add risk for malware operators, who will lose the ability to deploy new payloads if they lose the private key. Notably, earlier studies have shown the feasibility of leveraging C&C backend surveillance to retrieve credentials [16], [36], [37]. Should incident responders obtain the encryption key pair, they can still deploy a remediation payload with ECHO's findings.

B. Generality

The methodology employed by ECHO can be adapted to address malware across various platforms through their payload deployment routine. Windows malware may download and run standalone executable binaries or DLL files [38]. Besides, Linux malware (like Mirai) can run remote shell commands and execute remote payloads [39]. Incident responders can construct a comparable pipeline using S2E [40], simulating the malware and identifying the routines through concolic execution. Similarly, this approach is directly applicable to malware crafted in scripting languages, such as Electron apps, which often exploit dynamic code to run remote payloads [41].

C. Program Analysis Limitations

Like all taint-analysis-based malware research, ECHO may produce a false negative result for malware that can thwart state-of-the-art taint analysis tools. As such, anti-tainting techniques would pose a problem for ECHO's model edge generation and in-vivo influence analysis components, which both rely on tracking the data flow between binary statements. We consider solving the anti-taint analysis evasion as an orthogonal problem. However, ECHO 's design is modular and can incorporate more robust taint-analysis tools when they are developed. Besides, particular malware may develop a long idle period before fetching the remote payload and thus cannot be effectively triggered by ECHO's force-execution technique. This may weaken ECHO's capability of generating dynamic vertices from the sandbox. Although this is an orthogonal challenge for ECHO, the solutions from several studies [42], [43] can be utilized to overcome the problem. We also consider this as a future work direction.

VII. Ethical Considerations

We limit our study to analyzing malware in an isolated sandbox and passively monitoring their network traffic. The development and evaluation of ECHO was done only within our testbed on devices owned by our research lab (§IV).

Regarding real-world deployment, ECHO is designed for law enforcement and authorized organizations with legal permission. US federal law enables authorized incident responders to seize botnet traffic and access malware on private devices under *Rule 41 of the Federal Rules of Criminal Procedure* [20]. Similar legal frameworks exist in other jurisdictions [44]. Following the precedent set by prior botnet takedown campaigns [21], [22], [45], the use of ECHO is an allowable and effective option for incident responders with legal permission. This legal protection has also been extended to private companies working on behalf of national governments. Holt et al. [46] documented the legal and cooperative challenges faced by law enforcement, which limits their ability to respond to cybercrime alone. In response, national governments often seek cooperation with private companies for malware takedown, such as Microsoft's TrickBot takedown [21] and public web services taking action against suspected malware accounts [16], [47]. Notably, the *Retadup* takedown [23] required collaboration between the French police department and Avast to globally distribute a remediation to victims. This exemplifies how experts under law enforcement's supervision can utilize ECHO correctly.

Ideally, incident responders must carefully evaluate the consequences and side effects of deploying a remediation payload. Incident responders should get users' consent before disrupting or uninstalling the malware from their device whenever possible. Legal frameworks exist for ECHO to aid authorities in fighting cybercrime by allowing them to respond surgically and minimize collateral damage.

VIII. Related Work

Botnet Analysis. JACKSTRAWS [6] identified C&C communications by building a behavior graph of malware's system calls and data exchange over network connections. While it can potentially detect payload downloading, system-call-based behavior graphs are too coarse-grained to identify a reusable payload deployment routine. Without statement-level data flow tracking, the existing technique cannot be directly augmented to identify payload deployment routines. Thus, when malware encodes variables between API calls, JACKSTRAWS's data-flow tracking will break.

Further, JACKSTRAWS is orthogonal to generating and deploying a remediation payload. ECHO performs fine-grained statement-level analysis to the malware to identify C&C backends and their payload deployment routine. Paleari et al. [12] proposed automatically deleting dropped malware binaries on infected devices, but requires incident responders to have access to the device for manual deployment. ECHO is inspired by this work and expands on it in two essential ways: 1) ECHO not only deletes dropped binaries but also provides customizable payloads to enable various remediation capabilities and 2) ECHO enables payload deployment by reusing the malware's payload deployment routines.

Botnet detection via machine learning-based approaches has been widely covered by recent research [48], [49]. For a takedown, prior work focus on detecting and remediating botnets through invalidating C&C backends [11], [50]–[56]. Studies [57], [58] extract Android apps' backends via symbolic execution but are limited by path explosion and malware evasion. Nadji et al. [7] proposed a DNS-level solution to find botnet C&C servers. Stone-Gross et al. [9] showcased a temporary real-world botnet seizing. Additionally, prior approaches for analyzing malware behavior [59]–[62] noted the challenge of extracting payload deployment routines and remediating remotely controlled malware. Distinctively, ECHO emphasizes the malware takedown via their embedded update mechanism, empowering incident responders to clean up malware without directly accessing infected devices.

Dynamic Code Loading Analysis. Poeplau et al. [63] introduced a static analysis to detect external code loading in Android apps. Qu et al. [64] examined malicious behaviors in dynamically loaded Android code. Zhou et al. [65] used heuristics to identify suspicious dynamic code loading, while Falsina et al. [66] proposed a code verification protocol to help secure legitimate dynamic code routines. DVa [67] solely identifies dynamic code loaded by Android malware with accessibility capabilities. Different from these, ECHO identifies RDCL routines that can be flipped by incident responders to remediate malware.

WebView Analysis. Both Yang et al. [68] and BabelView [69] automatically find JSI vulnerabilities in apps. ECHO is motivated by these works, but rather than using static analysis or Monkey [70], ECHO employs hybrid data-flow analysis for capability profiling. Lee et al. [71] proposed a static analysis to fingerprint hybrid apps. Tang et al. [72] performed a comprehensive binary analysis of WebView-based malware. These works pinpoint malicious behavior in hybrid apps, which is complementary to ECHO's goal of reusing JSI routines to remediate malware.

IX. Conclusion

ECHO is an automatic framework to formalize and generate remediation solutions for remote-controlled malware. By deriving the formal model for the malware's internal payload deployment routines, ECHO enables the payload's capability analysis and remediation payload generation. Utilizing a hybrid technique, ECHO uses analysis of the malware to aid incident responders with legal authorizations in the cleanup of infected devices. With 702 samples, we demonstrated that ECHO empowers incident responders to rapidly remediate malware with a 74.50% takedown rate.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and feedback. We also thank our collaborators at Netskope for their support, insights, and suggestions throughout this research. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179 and N00014-23-1-2073; the National Science Foundation (NSF) under grant 2143689; and the Defense Advanced Research Projects Agency (DARPA) under contract N66001-21-C-4024. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors and collaborators.

References

[1] *Microsoft attempts takedown of global criminal botnet*, https://apnews.com/article/technology-malware-elections-crime-cybercrime-913ee5d56affa97fc5d9c639c4a284ab, [Accessed: 2023-10-08].

[2] *Massive 'botnet' revives after takedown*, https://www.cbsnews.com/news/massive-botnet-revives-after-takedown/, [Accessed: 2024-09-04].

[3] *Pcs still infected with andromeda botnet malware, despite takedown*, https://www.zdnet.com/article/pcs-still-infected-with-andromeda-botnet-malware-despite-takedown/, [Accessed: 2023-10-12].

[4] *FBI tackles DNSChanger malware scam*, https://www.cnet.com/news/fbi-tackles-dnschanger-malware-scam/, [Accessed: 2023-10-12].

[5] *It's hard to keep a big botnet down: TrickBot sputters back toward full health*, https://www.cyberscoop.com/trickbot-status-microsoft-cyber-command-takedown/, [Accessed: 2023-10-08].

[6] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "JACKSTRAWS: picking command and control connections from bot traffic," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[7] Y. Nadji, M. Antonakakis, R. Perdisci, D. Dagon, and W. Lee, "Beheading hydras: Performing effective botnet takedowns," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[8] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and analysis of Hajime, a peer-to-peer IoT botnet," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[9] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[10] *Trickbot gets updated to survive takedown attempts*, https://www.securityweek.com/trickbot-gets-updated-survive-takedown-attempts, [Accessed: 2020-12-01].

[11] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "SoK: P2PWNED - modeling and evaluating the resilience of peer-to-peer botnets," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2013.

[12] R. Paleari, L. Martignoni, E. Passerini, D. Davidson, M. Fredrikson, J. T. Giffin, and S. Jha, "Automatic generation of remediation procedures for malware infections," in *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[13] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "TARDIS: rolling back the clock on CMS-targeting cyber attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, Virtual Conference, May 2020.

[14] Z. Shan, X. Wang, and T. Chiueh, "Malware clearance for secure commitment of OS-level virtual machines," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 2, pp. 70–83, Mar. 2013.

[15] T. Hamed, R. Dara, and S. C. Kremer, "Chapter 6 - intrusion detection in contemporary environments," in *Computer and Information Security Handbook*, 3rd edition, Boston, MA, 2017, pp. 109–130.

[16] J. Fuller, R. P. Kasturi, A. Sikder, H. Xu, B. Arik, V. Verma, E. Asdar, and B. Saltaformaggio, "C3PO: Large-scale study of covert monitoring of C&C servers via over-permissioned protocol infiltration," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

[17] D. Bradbury, "Fighting botnets with sinkholes," *Network Security*, vol. 2012, no. 8, pp. 12–15, Aug. 2012.

[18] *Malrhino android banking trojan*, https://www.enigmasoftware.com/malrhinoandroidbankingtrojan-removal/, [Accessed: 2023-10-12].

[19] *Outbreak of android trojan xHelper malware appears to be triggered by google play itself*, https://hothardware.com/news/xhelper-malware-appears-to-be-triggered-by-google-play, [Accessed: 2023-10-12].

[20] *Recent botnet takedowns allow u.s. government to reach into private devices*, https://www.lawfaremedia.org/article/recent-botnet-takedowns-allow-u.s.-government-to-reach-into-private-devices, [Accessed: 2024-06-17].

[21] *TrickBot botnet survives takedown attempt, but Microsoft sets new legal precedent*, https://www.zdnet.com/article/trickbot-botnet-survives-takedown-attempt-but-microsoft-sets-new-legal-precedent/, [Accessed: 2023-10-12].

[22] *Google remotely removes apps from android phones for security reasons*, https://gizmodo.com/google-remotely-removes-apps-from-android-phones-for-se-5572510, [Accessed: 2022-05-15].

[23] *Police hijack a botnet and remotely kill 850,000 malware infections*, https://techcrunch.com/2019/09/01/police-botnet-takedown-infections/, [Accessed: 2023-10-08].

[24] *Xposed*, https://forum.xda-developers.com/f/xposed-general.3094/, [Accessed: 2022-09-20].

[25] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, *mitmproxy: A free and open source interactive HTTPS proxy*, https://mitmproxy.org/, [Version 9.0], 2010.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[27] *Virustotal*, https://www.virustotal.com/gui/, [Accessed: 2023-10-12].

[28] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Z. Snow, F. Monrose, and M. Antonakakis, "The circle of life: A large-scale study of the IoT malware lifecycle," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[29] S. Sebastián and J. Caballero, "AVClass2: Massive malware tag extraction from AV labels," in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, Virtual Conference, Dec. 2020.

[30] *Trojan:Android/SmsSend*, https://www.f-secure.com/v-descs/trojan-android-smssend.shtml, [Accessed: 2023-10-17].

[31] *Web API design best practices - azure architecture center*, https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design, [Accessed: 2022-02-04].

[32] *APKiD*, https://github.com/rednaga/APKiD, [Accessed: 2022-09-20].

[33] *Cisco Talos Intelligence Group - Comprehensive Threat Intelligence*, https://www.talosintelligence.com, [Accessed: 2022-04-15].

[34] *New action to combat ransomware ahead of U.S. elections*, https://blogs.microsoft.com/on-the-issues/2020/10/12/trickbot-ransomware-cyberthreat-us-elections/, [Accessed: 2022-02-05].

[35] *Russian security firm sinkholes part of the dangerous meris ddos botnet*, https://therecord.media/russian-security-firm-sinkholes-part-of-the-dangerous-meris-ddos-botnet, [Accessed: 2023-10-08].

[36] D. Andriesse, C. Rossow, and H. Bos, "Reliable recon in adversarial peer-to-peer botnets," in *Proceedings of the 2015 Internet Measurement Conference (IMC)*, Tokyo, Japan, Oct. 2015.

[37] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[38] *New DDoS botnet malware infecting windows, linux, and IoT devices*, https://cybersecuritynews.com/ddos-botnet-malware-infecting-windows/, [Accessed: 2023-10-08].

[39] *Mirai botnet attack IoT devices via cve-2020-5902*, https://www.trendmicro.com/en_us/research/20/g/mirai-botnet-attack-iot-devices-via-cve-2020-5902.html, [Accessed: 2022-05-15].

[40] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[41] *Electron bot malware is disseminated via Microsoft's official store and is capable of controlling social media apps*, https://resources.infosecinstitute.com/topics/malware-analysis/electron-bot-malware-is-disseminated-via-microsofts-official-store-and-is-capable-of-controlling-social-media-apps/, [Accessed: 2023-10-08].

[42] X. Wang, Y. Yang, and S. Zhu, "Automated hybrid analysis of android malware through augmenting fuzzing with forced execution," *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2768–2782, Dec. 2019.

[43] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International World Wide Web Conference (WWW)*, Perth, Australia, Apr. 2017.

[44] J. Healey, N. Jenkins, and J. D. Work, "Defenders disrupting adversaries: Framework, dataset, and case studies of disruptive counter-cyber operations," in *Proceedings of the 12th International Conference on Cyber Conflict (CyCon)*, Virtual Conference, May 2020.

[45] *World's most dangerous malware emotet disrupted through global action*, https://www.europol.europa.eu/newsroom/news/world%E2%80%99s-most-dangerous-malware-emotet-disrupted-through-global-action, [Accessed: 2023-10-12].

[46] T. J. Holt, "Regulating cybercrime through law enforcement and industry mechanisms," *The Annals of the American Academy of Political and Social Science*, vol. 679, no. 1, pp. 140–157, Sep. 2018.

[47] M. Yao, J. Fuller, R. P. Sridhar, S. Agarwal, A. K. Sikder, and B. Saltaformaggio, "Hiding in plain sight: An empirical study of web application abuse in malware," in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[48] T. A. Tuan, H. V. Long, L. H. Son, R. Kumar, I. Priyadarshini, and N. T. K. Son, "Performance evaluation of botnet DDoS attack detection using machine learning," *Evolutionary Intelligence*, vol. 13, no. 2, pp. 283–294, Jun. 2020.

[49] Y. N. Soe, Y. Feng, P. I. Santosa, R. Hartanto, and K. Sakurai, "Machine learning-based IoT-botnet attack detection with sequential architecture," *Sensors*, vol. 20, no. 16, p. 4372, Aug. 2020.

[50] H. Kato, S. Haruta, and I. Sasase, "Android malware detection scheme based on level of SSL server certificate," in *Proceedings of the 2019 IEEE Conference and Exhibition on Global Telecommunications (GLOBECOM)*, Waikoloa, HI, Dec. 2019.

[51] S. H. Mousavi, M. Khansari, and R. Rahmani, "A fully scalable big data framework for botnet detection based on network traffic analysis," *Information Sciences*, vol. 512, pp. 629–640, Feb. 2020.

[52] Z. Zhou, L. Yao, J. Li, B. Hu, C. Wang, and Z. Wang, "Classification of botnet families based on features self-learning under network traffic censorship," in *Proceedings of the 3rd International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC)*, Shanghai, China, Oct. 2018.

[53] N. Koroniotis, N. Moustafa, E. Sitnikova, and J. Slay, "Towards developing network forensic mechanism for botnet activities in the iot based on machine learning techniques," in *Proceedings of the 9th International Conference on Mobile Networks and Management (MONAMI)*, Melbourne, Australia, Dec. 2017.

[54] S. Almutairi, S. Mahfoudh, S. Almutairi, and J. S. Alowibdi, "Hybrid botnet detection based on host and network analysis," *Journal of Computer Networks Communications*, vol. 2020, 9024726:1–16, Jan. 2020.

[55] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul. 2008.

[56] E. Bou-Harb, M. Debbabi, and C. Assi, "Big data behavioral analytics meet graph theory: On effective botnet takedowns," *IEEE Network*, vol. 31, no. 1, pp. 18–26, Jan. 2017.

[57] C. Zuo and Z. Lin, "SMARTGEN: exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International World Wide Web Conference (WWW)*, Perth, Australia, Apr. 2017.

[58] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, "The betrayal at cloud city: An empirical analysis of cloud-based mobile backends," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[59] R. P. Kasturi, J. Fuller, Y. Sun, O. Chabklo, A. Rodriguez, J. Park, and B. Saltaformaggio, "Mistrust plugins you must: A large-scale study of malicious plugins in wordpress marketplaces," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[60] O. Alrawi, M. Ike, M. Pruett, R. P. Kasturi, S. Barua, T. Hirani, B. Hill, and B. Saltaformaggio, "Forecasting malware capabilities from cyber attack memory images," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[61] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Conference, Feb. 2021.

[62] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[63] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in android applications," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[64] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. D. Riley, "DyDroid: Measuring dynamic code loading and its security implications in android applications," in *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, Jun. 2017.

[65] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[66] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab 'n run: Secure and practical dynamic code loading for android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, Dec. 2015.

[67] H. Xu, M. Yao, R. Zhang, M. M. Dawoud, J. Park, and B. Saltaformaggio, "Dva: Extracting victims and abuse vectors from android accessibility malware," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[68] G. Yang, J. Huang, and G. Gu, "Automated generation of event-oriented exploits in android hybrid apps," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[69] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the impact of code injection attacks in mobile webviews," in *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Crete, Greece, Sep. 2018.

[70] *Ui/application exerciser monkey*, https://developer.android.com/studio/test/other-testing-tools/monkey, [Accessed: 2023-10-17].

[71] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, Singapore, Sep. 2016.

[72] Z. Tang, J. Zhai, M. Pan, Y. Aafer, S. Ma, X. Zhang, and J. Zhao, "Dual-force: Understanding webview malware via cross-language forced execution," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sep. 2018.

Appendix A
Android-Specific Vertex And API Mapping

We present the vertex and system API mapping targeting Android-specific platform in Table VIII.

TABLE VIII: Android-Specific API And Vertex Mapping

Class Name	Method Name	Vertexes
java.net.URLConnection	connect	v^f_{req}
java.net.URL	openStream	v^f_{req}, v^f_{res}
java.net.URLConnection	getContent	v^f_{res}
java.net.URLConnection	getInputStream	v^f_{res}
okhttp3.Call	execute	v^f_{req}, v^f_{res}
okhttp3.Call	enqueue	v^f_{req}
okhttp3.Response	body	v^f_{res}
retrofit2.Call	execute	v^f_{req}, v^f_{res}
retrofit2.Call	enqueue	v^f_{req}
retrofit2.Response	body	v^f_{res}
com.android.volley.RequestQueue	add	v^f_{req}
com.android.volley.Response.Listener	onResponse	v^f_{res}
org.apache.http.client.HttpClient	execute	v^f_{req}, v^f_{res}
org.apache.http.HttpResponse	getEntity	v^f_{res}
java.io.FileOutputStream	write	v^l_{fw}
java.io.BufferWriter	write	v^l_{fw}
java.io.PrintWriter	println	v^l_{fw}
java.io.DataOutputStream	writeInt	v^l_{fw}
java.io.DataOutputStream	writeUTF	v^l_{fw}
java.io.RandomAccessFile	writeUTF	v^l_{fw}
java.nio.file.Files	write	v^l_{fw}
java.io.DataInputStream	readInt	v^l_{fr}
java.io.DataInputStream	readUTF	v^l_{fr}
java.io.RandomAccessFile	readUTF	v^l_{fr}
java.nio.file.Files	readAllLines	v^l_{fr}
java.io.DataInputStream	readInt	v^l_{fr}
java.io.FileInputStream	read	v^l_{fr}
android.content.res.AssetManager	open	v^l_{fr}
java.io.BufferedReader	read	v^l_{fr}
java.io.BufferedReader	readLine	v^l_{fr}
android.util.Base64	decode	v^l_{dec}
org.json.JSONObject	<constructor>	v^l_{dec}
com.google.gson.Gson	fromJson	v^l_{fr}, v^l_{dec}
javax.xml.parsers.DocumentBuilder	parse	v^l_{fr}, v^l_{dec}
java.util.zip.ZipInputStream	read	v^l_{dec}
java.util.zip.GZIPInputStream	read	v^l_{dec}
javax.crypto.Cipher	doFinal	v^l_{dec}
N/A	XOR operation	v^l_{dec}
java.lang.String	substring	v^l_{seg}
java.lang.String	split	v^l_{seg}
java.lang.String	append	v^l_{seg}
org.json.JSONObject	getString	v^l_{seg}
org.json.JSONObject	getInt	v^l_{seg}
com.google.gson.JsonObject	get	v^l_{seg}
org.w3c.dom.Document	getElementsByTagName	v^l_{seg}
org.xmlpull.v1.XmlPullParser	nextText	v^l_{seg}
java.security.MessageDigest	digest	v^l_{verify}
javax.crypto.Mac	doFinal	v^l_{verify}
java.security.Signature	verify	v^l_{verify}
java.util.zip.Checksum	update	v^l_{verify}
java.util.zip.Checksum	update	v^l_{verify}
android.webkit.WebView	loadUrl	v^f_{req}, v^f_{res} v^e_{sce}
android.webkit.WebView	loadData	v^e_{sce}
android.webkit.WebView	loadDataWithBaseURL	v^e_{sce}
java.lang.ClassLoader	defineClass	v^e_{bcl}
dalvik.system.DexClassLoader	<constructor>	v^l_{fr}, v^e_{bcl}
dalvik.system.DexClassLoader	loadClass	v^e_{bcl}
dalvik.system.InMemoryDexClassLoader	<constructor>	v^e_{bcl}
java.lang.reflect.Method	invoke	v^e_{exe}

Appendix B
Dynamic Vertex Generating Sandbox

Given a malware sample as input, ECHO aims to detect suspicious payload deployment behaviors as the vertexes defined in §III-A. While static analysis can identify specific API calls from decompiled binaries, it is limited to resolving the parameters for each API. Thus, ECHO employs dynamic execution to monitor suspicious API calls.

ECHO performs the forced execution at the Android application component level to automate malicious behavior triggering. These components, foundational to Android apps, include `Activity`, `Service`, `Broadcast Receiver`, and `Content Provider`. Their lifecycle methods can serve as the entry point of malicious code. For example, with the fake Youku malware (see §II), ECHO initializes the malware and triggers all UI elements in each Component to trigger the remote code execution behavior in the `onCreate` method of `WelcomeActivity`. For Activities, that are highly interactive, ECHO's Execution Manager traverses through their lifecycle.

Figure 3 illustrates the Execution Manager (EM) navigating Activities of the malware. Before the malware's initialization in Step ②, ECHO instruments the zygote process and injects analysis code to hook critical Activity lifecycle methods like `onCreate` and `onResume` in ①. The analysis code spawns the same process of the analyzed malware during its initialization. As the entry Activity initializes, the rendered interactive UI elements' data is sent to the EM in Step ③. The EM maintains the status of the Activity stack and the corresponding UI data. Using a depth-first search strategy, the EM sequentially triggers each interactive UI element and its event handlers in ④. Upon one event getting triggered, the malware updates the current foreground Activity data with the EM again in ⑤. After all UI elements in the foreground Activity are engaged, the EM finishes the Activity in Step ⑥ and recursively navigates new foreground Activities. Post navigating all Activities, the EM concludes the analysis and terminates the malware in Step ⑦. For other components, ECHO initializes them and calls their lifecycle methods.

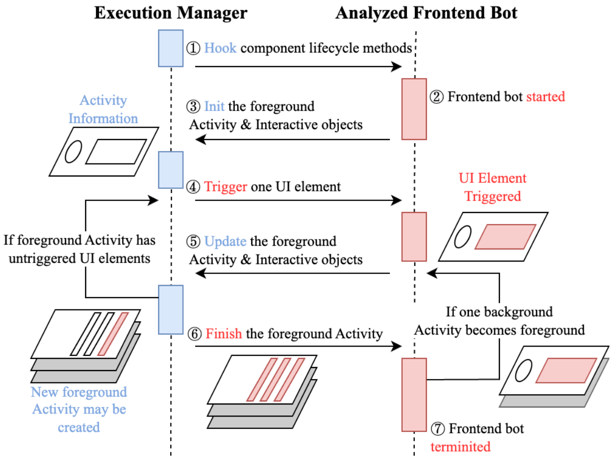

Execution Manager
Analyzed Frontend Bot
① Hook component lifecycle methods
② Frontend bot started
Activity Information
③ Init the foreground Activity & Interactive objects
④ Trigger one UI element
UI Element Triggered
If foreground Activity has untriggered UI elements
⑤ Update the foreground Activity & Interactive objects
If one background Activity becomes foreground
⑥ Finish the foreground Activity
New foreground Activity may be created
⑦ Frontend bot terminited

Fig. 3: Design of Execution Manager. Blue events happen at the Executor Manager's process and red events happen at the malware's process.

Algorithm 2: Multi-threading Stack Trace Generation

```
   // A Key-Value map to cache thread-external
      stack trace using thread id as the key.
1  const $cacheMap = $ new $Map()$;
   // Additional map to cache thread-external
      stack trace for thread pool case.
2  const $cacheJobMap = $ new $Map()$;
   // The method is called before hooked
      multi-threading methods.
3  Function BeforeMultiThreadingMethods($hookedMethod$):
4      switch $hookedMethod$ do
           // When developer's code starts a
              customized thread
5          case $thread.Start$ do
               // Get current stack trace with
                  system API.
6              $externalST = getCurrentStackTrace()$;
               // Get the identifier of current
                  thread and cache the external
                  stack trace.
7              $cacheMap.put(thread.id, externalStackTrace)$;
8          end
           // When a target API that we need to
              generate the full stack trace for is
              called.
9          case targetMethod do
               // Concat the cached thread-external
                  stack trace with current
                  thread-internal stack trace.
10             $internalST = getCurrentStackTrace()$;
11             $externalST = cacheMap.get(thread.id)$;
12             $fullST = externalST.concat(internalST)$;
13             return $fullST$;
14         end
           // When thread pool receives a new job.
15         case ThreadPool.submitJob do
               // Collect current stack trace with
                  system API.
16             $externalST = getCurrentStackTrace()$;
               // Use job's identifier as the key to
                  cache thread-external stack
                  trace.
17             $cacheJobMap.put(job.id, externalST)$;
18         end
           // When a job is assigned to a thread in
              the pool.
19         case ThreadPool.executeJob do
               // As the actual thread is assigned,
                  sync the thread-external stack
                  trace to cacheMap.
20             $externalST = chcheJobMap.get(job.id)$;
21             $cacheMap.put(thread.id, externalStackTrace)$;
22         end
23     end
24 end
```

Context Information Extractor. Besides triggering malicious behaviors from the malware, ECHO's sandbox captures the context information associated with each API call to resolve the payload deployment routines. Specifically, ECHO meticulously logs method call arguments and gathers supplementary details essential for *deployment routine identification*. The indirect data, such as the file path derived from `File.write`, is gathered by examining the caller's object. Notably, certain private fields, being inaccessible directly, are retrieved using reflection. For instance, ECHO investigates the private field of the declaring class to pinpoint the `ClassLoader` and determine the loaded binary path.

Appendix C
Multi-threading Stack Trace Generation

Generally, with `Thread.getStackTrace` API, ECHO can hook any method and collect the stack trace. However, suppose the method is executed within a spawned thread. The directly collected stack trace will only log the method hierarchy within the thread, and no information from the caller thread is captured. To overcome this challenge, ECHO models various multi-threading scenarios and generates the multi-threading stack trace to capture cross-thread method call relationship.

The basic idea behind this module is that ECHO must capture and cache the stack trace of the caller thread when the spawned thread is initialized. Algorithm 2 presents ECHO's strategy for generating multi-threading stack trace at runtime. With a map from the thread and the thread-external stack trace (Line 1), when a custom thread is initialized, ECHO caches its thread-external stack trace (Lines 5-8). Later, when a hooked audited API is called, ECHO can retrieve the thread-external stack trace and combine it with the thread-internal stack trace to generate a full multi-threading stack trace (Lines 9-14).

Lines 15-22 of Algorithm 2 describe the scenarios in which the code is executed with a thread pool, multiple threads are managed not by the developer but by the Android system's infrastructure. Compared to a customized thread, the challenge comes from when the code is submitted as a job to the thread pool; the actual thread to execute the job is not determined. In this case, ECHO manages an additional map to cache the thread-external stack trace using the job id as the key (Line 2). When a job is submitted to a thread pool, the thread-external stack trace is cached in this new map (Lines 15-18). As the job is executed in the thread pool, the cache is then transferred from the job-keyed map to the thread-keyed map (Lines 19-22).

On the Android platform, we model both simple thread cases and thread-pool cases (i.e., `Handler`, `Executor`, `ExecutorService`). For the `ThreadPoolExecutor` case, which implements a thread pool and takes `Runnable` object as the job, ECHO handles the case with an additional strategy. Based on the fact that the thread pool implements a First-In-First-Out sequence to execute `Runnable` object, instead of a job-keyed map, ECHO maintains a queue to sync with the internal job queue. The queue pushes the thread-external stack trace when a job is scheduled and pops one when a job is executed.

Appendix D
Dynamic Trace-Guided Data Flow Analysis

Traditional static taint analysis faces challenges from the *Override* feature, which is a common practice in Java and other modern programming languages. This technique allows developers to redefine methods from parent-class methods within child classes. As a result, static data flow becomes inaccurate, as the method identified in the binary may be overridden and thus not the one actually called. For example, as static analysis detects a call to `Runnable.run`, traditional taint analysis, such as FlowDroid [26], can raise many false alarms by mapping all potential `run` method implementations.

ECHO integrates insights from dynamic sandboxing to guide static data flow analysis, particularly addressing the caller-callee matching problem. ECHO concurrently logs call stack traces for methods that commonly use the `Overriding` feature as the sandbox navigates through the frontend bot. For instance, when there's a call to the `Runnable.run` interface, ECHO logs the multi-threading

TABLE IX: Validation Result for Remote Dynamic Code Loading Detection & Remediation.

Family	#Sample		#DCL Routines [1]					#C&C Backends				
	Tot.	w/R.	GT	Gl	TP	FP	FN	GT	Gl	TP	FP	FN
hiddenapp	6	4	2	2	2	0	0	4	4	4	0	0
fab[2]	3	2	1	1	1	0	0	2	2	2	0	0
hiddenads	3	2	1	2	1	1	0	1	2	1	1	0
downloader	2	2	2	2	2	0	0	2	2	2	0	0
skymobi	2	1	1	2	1	1	0	1	2	1	1	0
grayware	2	2	2	1	1	0	1	2	1	1	0	1
smspay	1	0	0	0	0	0	0	0	0	0	0	0
shedun	1	1	1	1	1	0	0	1	1	1	0	0
Total	20	14	10	11	9	2	1	13	14	12	2	1

[1] Columns 4-8 measure the number of unique RDCL routines.
[2] fakeAdBlocker.
The routines can be implemented by multiple variants.

stack trace, pinpointing the actual method that gets executed. After the sandboxing finishes, ECHO compiles all the logged calls and their multi-threading stack traces. This consolidated data is then supplied to the static data flow analysis, ensuring a more accurate and informed analysis.

ECHO combines the strengths of both static and dynamic analysis to enhance the accuracy of taint analysis. Initially, ECHO builds a complete Control Flow Graph (CFG) with decompiled binaries. Next, ECHO traverses the CFG to pinpoint sink and source APIs. For methods having overriding implementations, ECHO first looks up the dynamic logs and searches for any call log that corresponds to the method under analysis. To search for the match, ECHO leverages the multi-threading stack traces, method signatures, and call statement line numbers to align static method signatures from CFG with dynamically captured method calls. If a match is found, ECHO confidently proceeds with the data flow analysis for that specific method, ensuring accuracy. Otherwise, ECHO adopts a strategy similar to the state-of-the-art approach by matching all candidates. As the output, to identify data dependency between v^f_{res} and v^e_{exe} vertexes, ECHO outputs d for each vertex pair.

Appendix E
Additional Validation

Validation Dataset. In these experiments, we aim to validate ECHO's identification of payload deployment routines. We randomly selected 20 samples from the 1,057 sandbox-filtered dataset (detailed in §IV). We manually reverse-engineered these samples to derive ground truth. Recall from §IV that this dataset contains samples executed in the sandbox but does not have any payload deployment routines. We include these to verify that ECHO produces few false positives or false negatives.

RDCL Validation. Table IX presents the result of ECHO's RDCL routine identification. Column 1 shows the malware's family based on the *VirusTotal* scan report and AVClass2 [29] labeling. Columns 2-3 show the total number of malware samples and samples with RDCL routines. Note that because variations in malware implementation within a single family, the availability of RDCL routines may differ. Columns 4-8 and 9-13, respectively, show the number of *unique RDCL routines* and the number of *C&C backends* hosting the Java payload for each malware family, broken down into ground truth (GT) based on the manual investigation, ECHO's detection results (Gl), True Positive (TP), False Positive (FP), and False Negative (FN) results. For example, for *hiddenapp* family with six variants (Row 1), our manual investigation (GT) confirmed that four out of six samples share two different RDCL routines that fetch the payloads from four different backends, and it matches ECHO's results.

TABLE X: Validation Result for JSI-based Remediation.

Family	#Samples		#JSI Routines [1]					#C&C Backends				
	Total	w/R	GT	Gl	TP	FP	FN	GT	Gl	TP	FP	FN
spyagent	5	1	2	2	2	0	0	1	1	1	0	0
smspay	5	0	0	0	0	0	0	0	0	0	0	0
resharer	3	1	1	1	1	0	0	1	1	1	0	0
nimda	1	1	2	2	2	0	0	2	2	2	0	0
grayware	3	1	2	1	1	0	1	1	1	1	0	0
fakeinst	2	0	0	0	0	0	0	0	0	0	0	0
metasploit	1	1	1	1	1	0	0	1	1	1	0	0
Total	20	5	8	7	7	0	1	6	6	6	0	0

[1] Columns 4-8 measure the number of unique JSI routines.
The routines can be implemented by multiple variants.

ECHO successfully recovered nine unique RDCL routines out of 10 ground truth routines (90%), with one routine missing from the *grayware* family. Also, ECHO accurately identified 12 out of 13 backends (92.31%) with two FP cases from *hiddenads* and *skymobi* families. Our security researcher manually investigated the reason for the FN and the FP cases. The missed routine (FN) is caused by the malware having a dead backend, and the payload should be embedded in a JSON file. Although ECHO tries to respond to the request with a dummy binary when the backend gives no response, the frontend receives a payload in the unexpected format and thus crashes. Thus, ECHO missed the payload execution behaviors to identify the routine and the C&C backends. For two FP identifications, we identified that all routines are caused by the over-tainting problem when generating the formal model, which is a rare case.

JSI Validation. Table X presents ECHO's results for JSI candidates. Column 1 shows the malware's family and Columns 2-3 show the sample count and the numbers of samples with valid JSI routines for each family. Columns 4-8 and 9-13 separately show the numbers of *unique JSI routines* with remediation capabilities and the number of detected C&C backends hosting the JS payloads. Among 20 samples, we manually identified eight JSI routines from five samples.

ECHO successfully detected seven of them with one missing JSI entry point, due to the data flow tainting receiving an out-of-memory error with an oversize generated CFG. Also, ECHO successfully identified all six backends that the WebView loaded the payloads from. Compared with payload fetching data flow tracking, we find that JSI methods are implemented more straightforwardly and come with smaller CFG, which enables ECHO with a higher success rate. The validation also implies that as the routine is identified, ECHO can easily identify the C&C backend that the frontend tries to fetch payload from.

Conclusion. Overall, ECHO identifies 19 out of 40 samples with payload deployment routines. ECHO achieves an 88.89% accuracy in identifying 16 out of 18 routines in the validation.

Appendix F
Artifact Appendix

A. Abstract

This artifact includes a code repository and supporting documentation for ECHO. ECHO is an automated malware forensics pipeline designed to extract payload deployment routines and generate remediation payloads to disable or remove remote-controlled malware from infected devices. ECHO enhances malware takedown efforts by leveraging the malware's built-in update mechanisms to distribute crafted remediation payloads, thereby enabling the covert and timely removal of bots. The pipeline supports Android malware analysis and facilitates actions ranging from covertly warning users about malware infection to uninstalling the malware. The repository contains all modules of ECHO, including dynamic sandboxing, hybrid data flow analysis, payload deployment routine modeling, in-vivo influence analysis, and remediation payload template generation.

B. Description & Requirements

The pipeline accepts a remotely-controlled Android malware sample (i.e., a `.apk` file) as input and processes it through all modules of ECHO. A running example with step-by-step guidance is provided to walk users through the core functionalities of ECHO.

1) Security, privacy, and ethical concerns: Because of the unique nature of malware analysis, ECHO is designed to interact actively with malware and its potentially live C&C servers during sandboxing and the identification of payload deployment routines. This interaction is necessary to counteract malware's anti-analysis techniques—such as packing, dynamic code loading, and sandbox probing—to reveal payload deployment schemes and enable further remediation.

To adhere to the artifact evaluation regulations, we prepared a neutralized malware sample with a simulated C&C infrastructure. This approach ensures that the artifact evaluation process raises no inherent security, privacy, or ethical concerns while substantiating the major claims of the paper. However, users can run active malware to verify ECHO's claims at their own risk.

2) Artifact Access: The artifact is available with DOI: 10.5281/zenodo.14210565[4]. We provide the source code for each module, along with compiled `jar` and `apk` files, to facilitate a quick setup of ECHO's dynamic sandboxing and data flow analysis. Additionally, we offer a demo video[5] to showcase the pipeline execution and remediation deployment for user reference.

3) Hardware Prerequisite: ECHO's dynamic sandboxing requires an OEM-unlocked Android mobile device running AOSP 9.0 (preferably a Google Pixel 3). Since this hardware is beyond the scope of artifact evaluation, we provide pre-collected results from the dynamic sandboxing module and a demo video for reference. In addition, a Linux machine with a single-core CPU and 8 GB of memory is required and has been tested.

4) Software dependencies: ECHO's dynamic sandboxing module is implemented as an *Xposed* plugin and an Android application serving as the on-device agent. The artifact enables users to set up this pipeline with ROM flashing and dynamic hooking framework installation instructions provided in the repository. The other modules of ECHO can be deployed on a desktop running Ubuntu 20.04.6 LTS. However, ECHO should function with any recent version of Ubuntu or any Debian 11 and newer (64-bit) systems.

Additionally, ECHO's data flow analysis is built on top of *FlowDroid* and *Soot*. The software dependency for running the compiled *jar* version of ECHO is Java Runtime Environment version 1.8.0. ECHO's graph generation and remediation template generation rely on *Python* (version 3.8 or higher). For further details, please refer to the `README` documentation in the GitHub repository.

C. Artifact Installation & Configuration

For basic deployment, users should install Java 1.8 and Python 3.8 on a desktop running Ubuntu 20.04.6 LTS to run the pre-compiled modules. However, additional dependencies and setup steps are documented for users who wish to build ECHO's modules from the source and set up the dynamic sandboxing module.

D. Experiment Workflow

The "Usage" section of `README` provides instructions for running ECHO against the malware sample known as fake Youku[6] as an example. We provide a one-step script (`script/start.sh`) that processes this binary through the entire pipeline and generates results from all modules. This is the same sample introduced in Section 2 of the paper. This malware possesses remote-control capabilities that allow it to execute arbitrary Linux shell code from the C&C server. To mitigate possible security and privacy risks, we verified that the C&C servers with which the sample communicates have been neutralized. We simulate its communication using a man-in-the-middle (MITM) proxy and host the original payload collected when the C&C server was active.

Users can set up the dynamic sandboxing module and collect the results. Alternatively, users may begin with the pre-collected dynamic sandboxing results provided. Subsequently, users should run payload deployment routine graph building, in-vivo influence analysis, and remediation payload generation modules in sequence. We provide scripts for running each module in the repository with the expected results. To this end, the experiment will encompass all three major malware analysis tasks presented in the paper: payload deployment routine modeling, in-vivo influence analysis, and remediation payload construction. This process will require approximately one hour of human effort and a maximum of one hour of computational time using the basic setup (i.e., utilizing pre-collected dynamic sandboxing results).

The experiment will produce outputs that reveal the following:

[4] URL: https://doi.org/10.5281/zenodo.14210565

[5] URL: https://www.youtube.com/@CyFILab

[6] SHA-256 hash: `5135210444ad90b3a0d5aa5bd64fb06fedae8b44d0b35a6f7e14be6128b476cf`, package name: *com.youku.phone*

1) **Modeled Payload Deployment Routines**. The results are generated by running the `run_graph_building_module.sh` script. In detail, the script will first run the data flow modules to capture static data flow between payload deployment-related APIs and output the results as `fl_fetching_output.json` and `fl_loading_output.json`. Next, it will generate a JSON file to represent the payload deployment routine graph. The JSON file contains a list of modeled payload deployment routines. For each routine, the modeled vertices are listed under the "vertices" field, which includes the modeled method signatures, stack traces, accessed file paths (if applicable), and other essential information for graph generation. The "edges" field in each routine object contains the connections between vertices and the mechanisms through which these edges are established (e.g., via data flow or shared file access). For the fake Youku malware, users will observe that it fetches a payload from the host *sdk1.itracker.cn*, unzips and decodes it, and loads the payload using WebView. We provide detailed guidance in the artifacts' `README` file on interpreting various outputs. This includes the results from dynamic sandboxes, the static data flow captured as intermediate outputs, and the final generated graph. Additionally, we include examples to illustrate how ECHO-generated routines model the malware's payload fetching, loading, and execution behaviors.
   We detail how to interpret the output from the dynamic sandboxes, the static data flow captured as intermediate outputs, and the final generated graph, including examples to illustrate how ECHO-generated routines model the malware's payload fetching, loading, and execution behaviors in the artifacts' `README` file.
2) **In-Vivo Influence Analysis**. This output represents how the payload can affect the malware and is produced by running the `run_invivo_influence_module.sh` script. ECHO enhances the graph-building results by adding an "influence" field for each routine. For a payload deployment routine that runs a Java binary, the influence will be a string indicating arbitrary binary code. In the case of the fake Youku malware, the influence of the payload includes a list of JavaScript Interface (JSI) methods that can be invoked from the payload, and APIs that can disrupt the malware's operation. Specifically, users will see that a `runCmd` JSI method can call the `Runtime.exec` method, enabling the execution of arbitrary Linux shell commands. We provide more details about interpreting the results in the artifact's `README`.
3) **Remediation Payload Template**. This is the result of `run_remediation_generation_module.sh` script and consists of a template file that incident responders can deploy for remediation purposes. For the fake Youku sample, the resulting remediation template calls the previously identified `runCmd` API within the JavaScript payload to both notify victims and delete the malware. This template is illustrated in Figure 1 of the paper. Additionally, we provide a demo video for the remediation process by deploying this remediation payload to malware samples through an MITM proxy.

We also included an `example_outputs` folder in the artifact for reference.

E. Customization

For each module, we enabled customizable parameters that allow users to run ECHO's pipeline on different setups, malware binaries, sink/source APIs, and platform versions. Besides, ECHO's prototype implements multiple modules that can be utilized or extended for other Android malware analysis tasks. We list some examples below:

src/DynamicSandboxingManager: A generic sandboxing tool that enables component-level force execution and passive API logging. The sandbox also allows the generating of multi-threaded stack traces for each logged function. See Appendices B and C of the paper for more details.

src/DataFlowAnalysis: A hybrid data flow analysis tool that can take execution traces from dynamic execution as additional input to perform static data flow analysis.

Additionally, the remediation payload generation module supports five types of in-vivo influence as specified in the paper. However, users can extend this module to support additional in-vivo influence APIs.