C^{2}SR: Cybercrime Scene Reconstruction for Post-mortem Forensic Analysis

Yonghwi Kwon[1], Weihang Wang[2], Jinho Jung[3], Kyu Hyung Lee[4], and Roberto Perdisci[3,4]
[1]University of Virginia, [2]University at Buffalo, SUNY, [3]Georgia Institute of Technology, [4]University of Georgia
yongkwon@virginia.edu, weihangw@buffalo.edu, jinho.jung@gatech.edu, {kyuhlee, perdisci}@uga.edu

Abstract*—Cybercrime scene reconstruction that aims to reconstruct a previous execution of the cyber attack delivery process is an important capability for cyber forensics (e.g., post mortem analysis of the cyber attack executions). Unfortunately, existing techniques such as log-based forensics or record-and-replay techniques are not suitable to handle complex and long-running modern applications for cybercrime scene reconstruction and post mortem forensic analysis. Specifically, log-based cyber forensics techniques often suffer from a lack of inspection capability and do not provide details of how the attack unfolded. Record-and-replay techniques impose significant runtime overhead, often require significant modifications on end-user systems, and demand to replay the entire recorded execution from the beginning. In this paper, we propose C^{2}SR, a novel technique that can reconstruct an attack delivery chain (i.e., cybercrime scene) for post-mortem forensic analysis. It provides a highly desired capability: interactable partial execution reconstruction. In particular, it reproduces a partial execution of interest from a large execution trace of a long-running program. The reconstructed execution is also interactable, allowing forensic analysts to leverage debugging and analysis tools that did not exist on the recorded machine. The key intuition behind C^{2}SR is partitioning an execution trace by resources and reproducing resource accesses that are consistent with the original execution. It tolerates user interactions required for inspections that do not cause inconsistent resource accesses. Our evaluation results on 26 real-world programs show that C^{2}SR has low runtime overhead (less than 5.47%) and acceptable space overhead. We also demonstrate with four realistic attack scenarios that C^{2}SR successfully reconstructs partial executions of long-running applications such as web browsers, and it can remarkably reduce the user's efforts to understand the incident.

I. Introduction

Exploiting software has become a non-trivial process of chaining multiple exploits in various software layers. This is because exploiting a single vulnerability is often not sufficient to launch a successful attack, avoiding various protection techniques such as ASLR. As a result, understanding the attack delivery process (i.e., how a security incident unfolds) is critical for attack attribution and identifying espionage. For example, details about the attack delivery processes can help reveal malicious actors and compromised entities.

The Association for Crime Scene Reconstruction (ACSR) [32] defines crime scene reconstruction (CSR) as "*the forensic science discipline that aims to gain explicit knowledge of the series of events that surround the commission of a crime using deductive and inductive reasoning, physical evidence, scientific methods, and their interrelationships.*" CSR is an invaluable component of post-mortem forensic analysis because it reconstructs crime scenes, providing a more intuitive understanding of the crime [15], [82].

In the context of cybercrime, a similar capability to CSR is highly desirable. Reconstructing an execution of the attack delivery process to gain knowledge of the series of cyber events, which we call *cybercrime scene*, for post-mortem forensic analysis can open various opportunities. Specifically, cyber forensic analysts can investigate the reconstructed execution via various analysis tools, including malware analysis tools and debuggers, to gain more knowledge of attacks.

Unfortunately, reconstructing a cybercrime scene (i.e., execution of the attack process) of modern applications for forensic analysis is challenging because (1) an exploit often happens during a long-running execution of a complex and concurrent application, requiring reconstruction of a large portion of attack irrelevant execution and (2) user-interactions on the reconstructed execution for forensic analysis (e.g., execution inspection) can cause new syscalls and instructions to be executed, interfering with execution reconstruction technique (e.g., record-and-replay techniques). For instance, in some web attacks, such as malvertising attacks [20], [73], [6], malicious payloads are often delivered through a chain of multiple network servers. Reconstructing the malicious payload delivery process from a long-running web browser execution and allowing forensic analysts to use debuggers and inspectors is desirable while difficult to achieve.

Existing post-mortem analysis techniques are not suitable to handle the above scenarios. Specifically, forensic analysis techniques [49], [35], [54], [51], [65], [69], [10], [50] analyze system events (e.g., syscall) to identify causal dependencies between system subjects (e.g., processes) and objects (e.g., files and network addresses), and generate causal graphs. However, they often suffer from the lack of inspection capability and are unable to provide details of the incident. While record-and-replay techniques can be used for post-mortem analysis as they can replay a recorded execution, they also do not fit well in our scenario. Specifically, recording the fine-grained program execution [80], [5], [39], [70] often imposes significant runtime overhead. Coarse-grained record-and-replay techniques [27], [42] focus on system-level events to reduce the overhead. However, they often require modifications (e.g., customized kernel) or hardware supports on end-user systems.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23128
www.ndss-symposium.org

More importantly, we realize that there are two critical capabilities required for cybercrime scene reconstruction: (1) *interactable* execution reconstruction to allow analysts to investigate the execution in greater detail and (2) partial execution reconstruction from a long execution.

First, many recent attacks exploit various languages (e.g., JavaScript and WebAssembly) to leave fewer traces behind. For instance, a script-based fileless attack [88] is challenging to understand only with the system- or instruction-level traces (e.g., syscalls or instruction traces). Even if a forensic analyst investigates a replayed execution of such an incident using existing record-and-replay techniques, there exists a semantic gap between the system-level execution and the malicious payload delivered via high-level script. It is desirable, if not necessary, to leverage debugging tools for those scripting languages and new technologies to unfold details of the attack. However, existing record-and-replay techniques do not allow attaching such additional software to replay execution because they make a replay execution divert from its recording.

Second, most existing record-and-replay techniques require to replay an entire recorded execution trace *from the beginning of the execution*, even if a forensic analyst only wants to reconstruct a certain part of the trace (e.g., a limited time window around a suspected security incident). In our context, this particularly limits the effectiveness of the techniques because an analyst often wants to investigate a partial execution of long-running applications such as web browsers and email clients, which often run for days.

In this paper, motivated by the definition of crime scene reconstruction (CSR), we propose an important cyber-forensic capability, Cybercrime Scene Reconstruction (or C^{2}SR), that aims to reconstruct an attack delivery process (or cybercrime scene), for post-mortem forensic analysis. To the best of our knowledge, C^{2}SR is the first practical technique that enables both *interactable* and *fine-grained partial execution* reconstruction. In other words, C^{2}SR allows a forensic analyst to reconstruct a partial execution of a single task (e.g., a single browser tab of a web browser) and interact with the reconstructed execution, using debugging tools that are *not part of the recording*. C^{2}SR records system calls with their arguments and then partitions the trace (i.e., the recorded system calls) by resources, which we call *resource-based execution partitioning* (§ IV-A). The key intuition of the resource-based execution partitioning is that each autonomous execution (e.g., browser tabs) accesses resources in a disjoint way (i.e., different browser tabs access separate sets of resources). Hence, partitioning an execution trace by resources essentially slices the execution between autonomous executions. In addition, C^{2}SR allows live interactions of a reconstructed execution for forensic investigation purposes, as long as the interactions do not cause resource accesses that cannot be reconstructed via our new concept, *consistent resource accesses*, that allow different yet reproducible resource accesses.

Our contributions are summarized as follows:

- We propose a new cyber forensic capability: cybercrime scene reconstruction. To the best of our knowledge, this is the first technique that enables an interactable partial execution reconstruction.
- We propose the novel concept of resource-based execution partitioning, along with practical resource reconstruction methods and algorithms for the cybercrime scene reconstruction. (§ IV-A and § IV-B).
- We develop and evaluate a prototype of C^{2}SR. The evaluation results show that C^{2}SR's partial execution is highly effective in practical forensic investigations that include long-running and complex real-world applications such as Firefox. C^{2}SR can reproduce security incidents by reconstructing less than 1% of the entire trace with reasonable recording and execution reconstruction overhead: less than 5.47% and 8.31% respectively).

Scope. We compare our work with other research in the area to draw a clear scope. Specifically, this research focuses on reconstructing the cybercrime scene (i.e., a partial execution that is directly related to the attack) for post-mortem forensic analysis. In particular, we focus on enabling user interactions with reconstructed execution so that a forensic analyst can utilize various debugging or forensic tools that did not exist at recording time. In contrast, record-and-replay techniques often aim to replay recorded executions faithfully and deterministically. They do not allow replay execution to diverge from the recording, and thus, additional debugging or forensic tools cannot be attached to the replay execution. C^{2}SR does not aim to provide a deterministic or faithful replay at the instruction-level. Instead, we design and implement a novel technique to reconstruct an interactable execution that is consistent with the recorded execution trace at the resource access level.

Assumptions and Limitations. C^{2}SR targets to reconstruct a specific incident from a program running multiple tasks (e.g., a web-browser opens multiple tabs) where each task accesses the resources in a disjointed way. C^{2}SR would not be effective if multiple tasks concurrently access the same resource in a non-deterministic way. Furthermore, reproducing concurrency bugs (i.e., executions that are sensitive to the resource access orders) is out of this paper's scope. Besides, we assume that all the system resources are accessed through syscalls, and C^{2}SR captures resource-accessing syscalls. Although it is uncommon in practice, if the resource is modified directly by the kernel without invoking syscalls, we cannot capture them. In terms of performance overhead, C^{2}SR's recording overhead is similar to other log-based techniques leveraging syscall or library hooking [49], [54], [51], [69], [10], [50], [92], [62].

II. Motivating Example

We use a realistic malicious advertisement attack (synthesized from real-world incidents [97], [26]) to demonstrate the effectiveness of C^{2}SR in a forensic investigation scenario. In this scenario, a victim uses Firefox to open multiple web pages in multiple tabs, and one tab loads a malicious website that downloads a malware (i.e., drive-by download).

A. Attack Scenario

The victim has nine browser tabs opened before it accesses the compromised website. Then, the victim opens a new tab (the 10th tab) and navigates to *www.forbes.com*, which happens to include a malicious online advertisement. The malicious ad executes JavaScript code and secretly launches a WebAssembly module in the user's browser. The WebAssembly module includes code that downloads a malware binary.

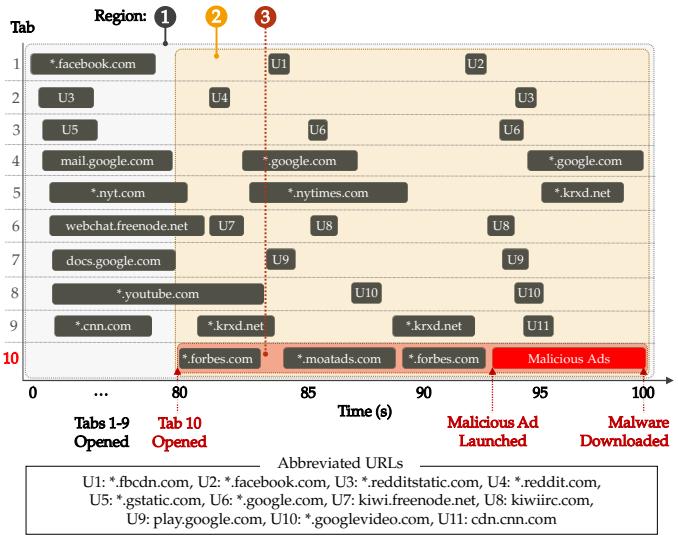


Fig. 1. Timeline of an execution of 10 browser tabs. The x-axis represents the time for the execution (in seconds) and the y-axis represents 10 different browser tabs. The abbreviated URLs (i.e., U1~U11) are shown in the bottom.

Timeline and Tabs during the Attack. Fig. 1 shows a timeline of Firefox during the incident. We visualize three different regions that include the entire execution of the browser from the beginning (1), the execution of all browser tabs during the malicious ad (2), and the execution of the browser tab containing malicious ad (3). Note that while the malicious ad executes, other tabs in the background are also loading contents (e.g., content updates) and, in our case, the executions of background tabs are longer than the exploited tab 10.

B. Goals and Scope

To understand how the attack unfolded in detail, C^{2}SR aims to *(1) reconstruct and reproduce the attack delivery process* (which occurs in tab 10), that is responsible for loading the malicious advertisement on *www.forbes.com* and delivering the malicious payload downloading the malware (essentially the red shaded region 3).

Moreover, for effective post-mortem forensic analysis, C^{2}SR aims to *(2) allow live user interactions for introspecting reconstructed executions* so that security analysts can use various forensic analysis tools to effectively investigate the reconstructed incident. Specifically, to understand a website's execution (including the malvertising campaign) which involves multiple software layers such as HTML, JavaScript, and WebAssembly where each of them has a unique semantic and runtime support, domain specific analysis tools for each layer [30], [25], [2] are particularly effective.

Identifying the Exploited Tab to Reconstruct. The recorded execution trace includes 10 browser tabs' executions. Among them, the forensic analyst identifies Tab 10 that creates and writes the malicious binary file. However, in our scenario, the analyst does not know how the website delivers a malicious payload that downloads the malware. Hence, she wants to reconstruct the entire execution of the tab to investigate.

C. C^{2}SR on the Motivating Example

1) Reconstructing the Attack: The forensic analyst uses C^{2}SR to reconstruct Tab 10 (*www.forbes.com*) to understand how the malicious payload that downloads the binary is delivered. The analyst focuses on discovering technical details of the attack delivery as well as identifying responsible entities (e.g., web servers) involved with the attack.

Starting to Reconstruct the Exploited Tab. To reconstruct the attack delivery process of Tab 10 (that navigated *www.forbes.com*), the analyst runs C^{2}SR with Firefox and provides the recorded execution trace as input. After Firefox is launched, the default start page is loaded. Then, the analyst types '*www.forbes.com*' in the address bar to initiate the execution reconstruction. When the browser connects to the domain (e.g., sending a DNS request), C^{2}SR hooks the API for the DNS request (e.g., `getaddrinfo()`) and detects that there is an autonomous task that starts from the network access to *www.forbes.com*, meaning that the task can be reconstructed. Then, it starts to reconstruct the recorded execution by redirecting resource accesses to reconstructed resources obtained from the execution trace.

Resource-based Execution Partitioning. The essence of C^{2}SR that makes an *interactable* partial execution reconstruction possible is the idea of resource based execution partitioning. It is based on the observation that individual partial executions from a long-running application mostly *access disjoint sets of resources*, meaning that each partial execution mostly accesses different resources from other executions, and rarely interfere with them. Intuitively, each browser tab is independent of other browser tabs. Hence, a partial execution can be obtained by *partitioning the execution trace by resources accessed during the partial execution*.

C^{2}SR partitions an execution trace, consisting of syscalls, by resource. Specifically, it first groups syscalls that access the same resource. For each group, all syscalls in the same group access the same resource. For example, suppose that a program receives contents from *www.forbes.com*. There will be a group for *www.forbes.com* and it includes all the syscalls that access (i.e., read and write) the *www.forbes.com* file exclusively. C^{2}SR reconstructs resources from the partitioned groups (of syscalls) on individual resources, resulting in *reconstructed resources*. Each reconstructed resource contains all the values and states observed during the recording and is capable of emulating the original resource (e.g., a web server). During reconstructed execution, C^{2}SR essentially hooks resource accessing syscalls to redirect them to access the reconstructed resources. To this end, a partial interactable reconstructed execution is created. A forensic analyst is allowed to interact with the reconstructed execution, including installing a new software for inspection, as long as it does not prevent C^{2}SR from reproducing *consistent* resource accesses. The in-depth inspection of the reconstructed execution help unfold details of the attack delivery process. More details about the definitions and limitations of a reconstructed execution are in § IV-A.

2) Investigating the Attack Delivery Process: In advanced attacks, simply reconstructing the execution of the attack is not sufficient to understand the attack. For example, in this motivating example, the analyst already knows that it downloads a malware binary. Reproducing the execution without allowing more in-depth inspections is not helpful. Instead, *how the malicious code that downloads the malware binary is delivered and executed* is worth investigating. This requires using debug-

ging tools (that often did not exist during recording) on the reconstructed execution. C^{2}SR allows the analyst to interact with the reconstructed execution so that the analyst can install and use additional debugging tools [29], [30] for investigation. In the following paragraphs, we show how analysis tools (that require user interactions) are used for investigating the motivating example.

Browser Extension to Find the Malicious Payload. Investigating web attacks is challenging due to the sheer number of loaded contents and executed code. In this motivation example, the forensic analyst only knows that the malicious code that downloads the malware is dynamically generated. Hence, she uses a Firefox extension called *Villain* [29] to monitor dynamically generated JavaScript code (e.g., via `eval()`). Note that the extension did not exist during the recording. The forensic analyst introduces the new extension for inspection. The extension allows the analyst to focus on the executed code that is already decoded and deobfuscated.

The analyst installs this extension and uses it to identify malicious payload generated and executed via `eval()` (B in Fig. 2). Note that existing record-and-replay techniques do not even allow installing a new extension that runs on top of a replayed execution, because the installation causes a number of new instructions and syscalls to be executed.

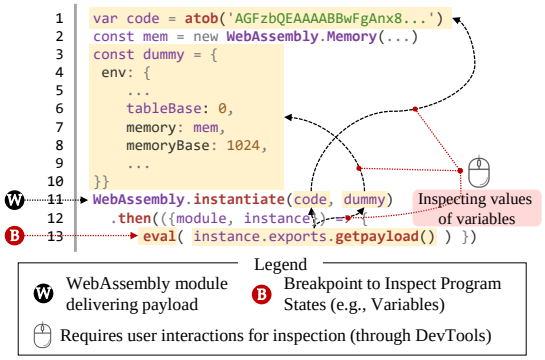


Fig. 2. Malicious Payload Delivery via WebAssembly and JavaScript.

Investigating the Attack Delivery Process. Fig. 2 shows the identified `eval()` that delivers the malicious payload. Then, the analyst leverages Firefox's default debugger (i.e., *DevTools* [30]) to set a breakpoint at line 13 (B), which executes the malicious payload (via `eval()`) returned from a WebAssembly module through `getpayload()` method. Note that the WebAssembly module is also dynamically instantiated. At line 11 (W), the module is created from the code and dummy objects, where the WebAssembly binary is encoded as base64-encoding at line 1. Since all these processes happen dynamically, leveraging *DevTools* to inspect the reconstructed WebAssembly program's code is particularly helpful. Further, the analyst sets the breakpoint at line 11 to understand how the WebAssembly module is created. By inspecting the arguments (e.g., `code` and `dummy`) and the resulting WebAssembly object, she finds out that it injects the `code` into the `dummy` object to create the WebAssembly module. Without the interactability of C^{2}SR, monitoring the reconstructed execution provides limited details regarding how the attack unfolds, particularly how the malicious payloads are transferred and generated secretly.

Summary. Post-mortem forensic investigation often requires the capability of reconstructing a partial execution from a long-running application. More importantly, as attacks become more sophisticated and stealthy (e.g., fileless attacks leveraging WebAssembly), detailed on-the-fly inspection of the reconstructed execution is highly desirable. In particular, allowing to install and use new plug-ins and debugging tools on the reconstructed execution significantly enhances forensic analysis capabilities. C^{2}SR enables an interactable partial execution reconstruction, providing a practical solution for post mortem forensic analysis.

TABLE I. LIMITATIONS OF EXISTING APPROACHES.

Technique	Partial Replay	Detail Inspection	Record Overhead	Mod. not Required	General Approach
Fine-grain Rec/Rep	No	Partially	High	Not required	Yes
Rec/Rep by System mod.	No	Partially	Low	Required	Yes
Replay Acceleration	Yes	Partially	N/A	N/A	Yes
Log-based Forensics	N/A	No	Low	Partially	Yes
Replay-based Forensics	No	No	Low	Required	Yes
Browser Replay	No	No	Low	Required	No
Browser Forensics	N/A	Partially	Low	Required	No
C^{2}SR	Yes	Yes	Low	Not required	Yes

* N/A: Not applicable.

D. Limitations of Existing Approaches

Table I summarizes the limitations of existing techniques in investigating attacks similar to the motivating example.

Record-and-replay. Fine-grain record-and-replay techniques record program instructions and/or shared memory access information to enable deterministic replay [80], [5], [39], [70]. Also, there exist approaches [27], [87], [91], [21] that modify applications, kernel, or hardware to enable a faithful replay. [37], [45] can replay a program with additional debugging code while they require recompilation/modification of the target program. Moreover, they do not allow additional debugging tools running together with the target application such as *DevTools*, limiting the applicability of the replayed execution. In general, they suffer from significant runtime and space overhead. DoublePlay [96] greatly reduces logging overhead by parallelizing the record-and-replay executions, however, it requires additional resources for the parallelized replay. In addition, traditional record-and-replay techniques replay an execution from the beginning of a recorded execution. In our motivating example, the entire execution of 10 browser tabs (the region 1) has to be replayed, leading to needless cost and effort to investigate irrelevant executions.

Replay Acceleration. Checkpointing techniques [93], [52], [91], [53] create checkpoints of the execution periodically or on particular events (e.g., process creation) during recording. A replay can be started from one of the checkpoints. However, they still need to replay nonessential concurrent tasks (e.g., in Fig. 1, the yellow shaded region 2 will be replayed if the checkpoint is created at 80 seconds, when the Tab 10 is created). Replay reduction technique [56] can reduce a replay log while retaining its ability to reproduce a failure. However, it requires source code annotation. Furthermore, replay acceleration techniques and faithful replay tools typically do not support interactable replay (e.g., debugger integration) as additional instructions and system calls invoked by debuggers are not seamlessly handled. REPT [22] is a reverse debugging technique to reproduce software failures by recovering program state (e.g., data values). It focuses on faithfully replaying

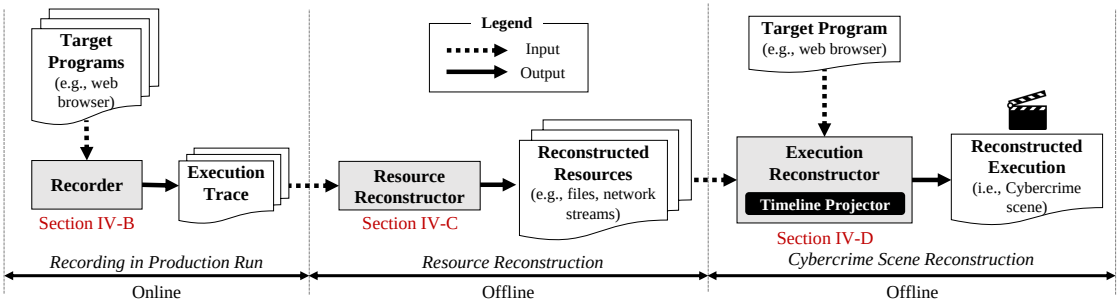


Fig. 3. Workflow of C^{2}SR. Shaded boxes represent components of C^{2}SR.

short execution that is immediately before the program crashes (e.g., less than 100K instructions in the paper's evaluation). As the authors mentioned in the paper, the data recovery accuracy decreases if the execution trace increases. In our scenario, we aim to reconstruct the entire attack delivery process that is longer than REPT can usually handle.

Forensic Analysis. Log-based forensic analysis techniques analyze system events (e.g., syscalls) and generate causal graphs [49], [35], [54], [51], [65], [69], [10], [50], [92], [62]. The main limitation of them is the lack of inspection capability. They focus on identifying causal relations between system subjects and objects but do not provide details of attack behaviors. In our example, they can identify the origin of the attack (e.g., IP addresses for *www.forbes.com* and the malicious advertisement) while they cannot reconstruct the execution of JavaScript and WebAssembly modules, failing to provide the details of the attack delivery process. Replay-based forensic analysis approaches [17], [67], [84], [28], [42], [43], [44] generate causal graphs by replaying the recorded execution log. However, they inherit the limitations of record-and-replay techniques (e.g., lack of partial replay capability and the requirement of system modification).

Brower-specific Approaches. Browser-level (or domain-specific) recording and replay techniques [71], [13], [16], [7], [66], [13] are effective in reproducing web- and web-application related executions. They allow replaying complicated web components such as JavaScript execution or user interactions with web applications. Enhanced browser logging [95], [11], [72], [57] can capture web-specific events to enable the investigation of web-based attacks. However, both of browser-specific techniques typically require browser instrumentation or extensions. Furthermore, they can only handle web attacks that completely unfold inside the browser.

III. System Overview

Fig. 3 shows a workflow of C^{2}SR, which consists of three phases: *Recording in Production Run*, *Resource Reconstruction*, and *Cybercrime Scene Reconstruction*.

Online Event Recording. C^{2}SR recorder logs system calls of a target process. Typically, one may log multiple processes in production run, as any of them might be exploited. C^{2}SR hooks APIs that invoke syscalls (via shared library and `LD_PRELOAD` trick) on the target program. It does not require modifications on target programs (e.g., instrumentation). The recorder generates an *execution trace* which is a sequence of executed system calls with arguments and timestamps.

Offline Trace Post-Processing. Given the execution trace, C^{2}SR resource reconstructor recovers states of resources (and resource content for each state) accessed during the recording. Then, it creates *reconstructed resources* from the recovered states and contents.

Offline Cybercrime Scene Reconstruction. C^{2}SR execution reconstructor takes the target program and the reconstructed resources as input. Then, it executes the target program and monitors all resource accesses at runtime. When the program tries to access resources that exist in the reconstructed resources, C^{2}SR redirects the accesses to them. C^{2}SR allows the reconstructed execution to be different at instruction and syscall levels, as long as they have consistent resource accesses with respect to the recorded execution. This design choice provides forensic analysts with the ability to interact with the reconstructed execution at replay time.

IV. Design

A. Concepts

C^{2}SR introduces a few concepts for partial and interactive execution reconstruction. Fig. 4 illustrates the concepts and differences between C^{2}SR and the existing approaches. Each box in Fig. 4 represents access to a particular resource where the color of the box indicates which resource is accessed.

Partial (Reconstructed) Execution. Fig. 4-(a) shows an execution trace without any partitioning. It is simply a sequence of system events (i.e., resource accesses in our context). Traditionally, a partial execution is often interpreted as a part of execution between a certain time period as shown in Fig. 4-(b). This definition of partial execution essentially includes all syscalls (i.e., it includes all different types of resource accesses) that happened between the beginning and the end of the time period (i.e., from T_{START} to T_{END} in Fig. 4-(b)).

The goal of C^{2}SR is to reconstruct an autonomous task. Hence, a *partial reconstructed execution* in our context, as shown in Fig. 4-(e), is defined as a reconstructed execution between two user-specified syscall instances, i.e., S_{BEGIN} and S_{END}, in an execution trace where the two syscalls represent the first and last syscalls of the partial execution. Typically, S_{BEGIN} is a syscall that initiates the entire task (e.g., `getaddrinfo()` to obtain IPs for a domain in a browser tab) and S_{END} is the one that delivers the attack (e.g., executing malicious payloads). Note that unlike the traditional (time-sliced) partial execution, our definition does not require including all syscalls between T_{START} and T_{END}. As long as the execution starting from S_{BEGIN} can reach to S_{END} without software faults (e.g., runtime errors), the partial execution is successful.

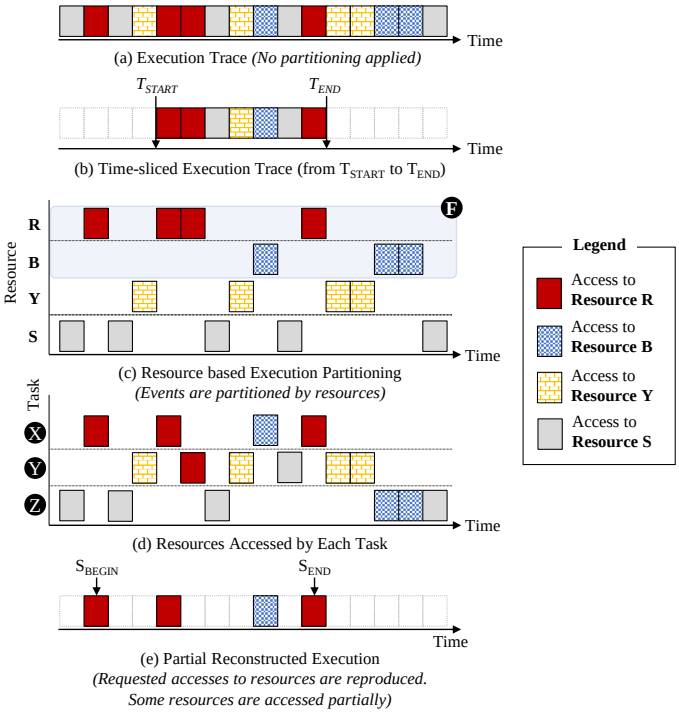


Fig. 4. Resource based Execution Partitioning Illustrated.

Resource based Execution Partitioning. The key enabling technique for partial execution reconstruction is resource-based execution partitioning. Given an execution trace, it essentially groups syscalls that access the same resource, as shown in Fig. 4-(c). Specifically, as there are four different resources (i.e., Resource R, B, Y, and S), syscalls (i.e., events illustrated as boxes) are partitioned into 4 groups. Fig. 4-(d) shows resource accesses for each task (e.g., a task corresponds to a browser tab in a web browser). In Fig. 4-(d), there are 3 tasks: Task X, Y, and Z. Observe that each task accesses multiple resources, and there are also cases that one resource (e.g., Resource R and B) is accessed by multiple tasks (e.g., multiple websites may access one server for different requests).

Recall that the goal of C^{2}SR is to reconstruct an execution of a task. A naive approach to reconstruct a recorded execution is to reproduce the recorded resource accesses of a given task. For example, to reconstruct Task X, one may try to reproduce all resource accesses to Resource R and B, marked as F. Unfortunately, this does not work because a resource can be accessed by multiple tasks (e.g., Resource R is accessed by Task X and Y). Reproducing resource accesses that belong to another task can break the execution reconstruction. For example, when the reconstructed task X accesses the third resource R, reproducing the access of the third access of resource R that belongs to the task Y can lead to an incorrect reconstruction. To this end, we propose a concept of *consistent resource access* that defines an execution accessing parts of resources needed for the execution reconstruction.

Reproducing Consistent Resource Access. A key difference between C^{2}SR and existing record-and-replay techniques is that C^{2}SR aims to reproduce *consistent results* of syscalls while existing techniques try to reproduce *faithful replay* of syscalls (e.g., including the exact order of syscalls). *Consistent results* in our context mean that results of resource accesses in a reconstructed execution are *semantically compatible* with its recorded resource accesses.

It relaxes two key restrictions that traditional record-and-replay techniques have. First, C^{2}SR allows *the order of reproduced syscalls to be different* from the recorded execution. Second, a reconstructed execution does not have to *reproduce every resource access* observed during recording. In other words, even if a reconstructed execution accesses parts of resources, the reconstructed execution is still considered successful as long as it does not access resources that are *not accessed during the recording*. A reconstructed execution can have additional syscalls as long as their resource accesses are *consistently reproducible* (e.g., reading the parts of resource already accessed is reproducible) or they do not access external resources. The relaxations make reconstructed executions *interactable*, meaning that additional syscalls caused by user interactions can be tolerated, so they do not lead to reconstruction failures. To this end, a reconstructed execution by C^{2}SR also tolerates non-determinism as long as it does not lead to unreproducible resource accesses (e.g., a new network connection that is never observed during recording).

Typically, user interactions for forensic investigation request existing resources accessed during the recording. For example, consider a scenario that an analyst uses a debugger to inspect a reconstructed execution. The analyst wants to examine parts of a suspicious file created by the reconstructed execution. To do so, the debugger may invoke a few new syscalls, to read the file, that were not observed during recording. While such new syscalls would make existing techniques fail, C^{2}SR can tolerate them because the new syscalls simply access the existing file's content that is already accessed during the recording. Hence, they can be easily reproduced by reaccessing the content.

B. Formal Definition of the Concepts Introduced by C^{2}SR

Trace	T ::=	$\overline{S}$
Syscall	S ::=	$< SysName,\ R,\ \mathbb{P}(C_{ARG}),\ \mathbb{P}(C_{RET}) >$
SyscallName	$SysName$ ::=	`open` | `read` | `write` | ...
ConcreteValue	C ::=	$\mathbb{Z}$
Resource	R ::=	$\mathbb{Z}$

Fig. 5. Definitions for execution reconstruction.

Definition Fig. 5 introduces definitions for recorded execution trace and executions reconstruction. Specifically, we define an execution trace (T) as a sequence of syscalls ($\overline{S}$). A syscall is defined as a tuple of a syscall name ($SysName$), a target resource handle (R), a set of its argument values ($\mathbb{P}(C_{ARG})$), and a set of its return values ($\mathbb{P}(C_{RET})$).

Resource based Execution Partitioning. C^{2}SR reproduces a *partial execution* from a resource-partitioned execution trace. To define partial execution reconstruction, we introduce two key concepts. First, we introduce definitions for the beginning and the end of a partial execution. Second, we focus on reproducing *consistent results of resource accesses* instead of faithful resource accesses required by existing approaches.

Concept 1: Partial Execution. A *partial reconstructed execution* is defined as a reconstructed execution between two user-specified syscall instances in an execution trace: the first and last syscalls of the partial execution.

• Definition 1 – S_{BEGIN} and S_{END}: We define the first syscall (S_{BEGIN}) and the last syscall (S_{END}) in a trace that indicates the beginning and the end of the reconstructed execution. Typically, S_{BEGIN} is the first syscall that accesses a key resource for the reconstructed execution. For instance, in our motivation example, S_{BEGIN} is the DNS request for *www.forbes.com* (i.e., `getaddrinfo()`), which should happen before any other network requests for the domain. S_{END} is often a syscall that an analyst wants to reproduce (e.g., creation of a suspicious file).

Concept 2: Reproducing Consistent Resource Access. A key difference between C^{2}SR and existing record-and-replay techniques is that C^{2}SR aims to reproduce *consistent results* of syscalls rather than identical results. *Consistent results* mean that (1) results of resource accesses in a reconstructed execution are *logically identical* to its recorded execution while (2) the order of syscalls can be different. For instance, if a `read()` on a file is observed during in the recording, the reconstructed execution should reproduce *consistent* values for the `read()` on the same file while it may allow syscalls that are independent to the file before the `read()`. To formally define consistent resource accesses, we introduce two definitions.

• Definition 2 – Resource-Partitioned Sub-Trace T_r (where r is a resource): We define T_r as a sub-trace on a resource r of the entire trace T. In other words, T_r is a sequence of syscalls operating on a resource r. For instance, if r is a file, T_r is a sequence of syscalls on r (e.g., `read(`r`)` and `write(`r`)`).

• Definition 3 – Resource Contents (RC): It is a mapping between a sub-trace T_r and a set of tuples where each tuple consists of a concrete value (C) of a resource's contents and its offset (O). It represents the concrete contents of a resource.

$$\text{RC} : T_r \mapsto \mathbb{P}(< C, O >) \text{ where } O \text{ is an offset } (\mathbb{Z}^+)$$

For instance, if a file f is accessed, RC is $T_f \mapsto \mathbb{P}(< B_i, i >)$ where B_i represents a byte value at an offset of i.

Definition of Successful Execution Reconstruction. We consider an execution reconstruction is successful if (1) a reconstructed execution correctly reproduces S_{BEGIN} and S_{END} and (2) all resource accesses between S_{BEGIN} and S_{END} are successfully and consistently reproduced.

Formal Definition. Let r_{c1}, r_{c2}, ..., r_{cn} be resources accessed during a recorded execution between S_{BEGIN} and S_{END}. T^c is an execution trace of the recorded execution. Let r_{p1}, r_{p2}, ..., r_{pn} be the corresponding reproduced resources accessed in a reconstructed partial execution. T^p is a partial execution trace to be reproduced. The partial resource-partitioned execution reconstruction is successful if the two following conditions are satisfied.

• *Condition 1.* S_{BEGIN} and S_{END} appear in T^p and the instance of S_{BEGIN} precedes the instance S_{END};

• *Condition 2.* $\forall r_{pi} \in R_p$, $\text{RC}(T^p_{pi}) \subseteq \text{RC}(T^c_{ci})$, where A $\subseteq$ B means A is equal to or a subset of B.

Summary. C^{2}SR partitions an execution by resources and reproduces *consistent results* of resource access. Unlike existing techniques that aim to faithfully replay a recorded execution (at instruction or system-event level), C^{2}SR allows two highly desired capabilities for forensic analysis: (1) a partial execution reconstruction and (2) an interactable reconstructed execution.

C. C^{2}SR *Recorder*

C^{2}SR recorder logs syscalls including their arguments and return values along with timestamps. If syscall arguments and returns contain values that may vary across executions (e.g., memory addresses), C^{2}SR abstracts them into the forms that do not vary across execution (e.g., offsets from the base addresses and filenames). As a result, corresponding syscalls between a recorded and reconstructed execution can be identified properly.

To log syscalls, we hook library calls that invoke syscalls (e.g., libc library calls). Logs are buffered on the memory and then written to the file system when it reaches a predefined threshold to minimize performance overhead caused by I/O for logging. The threshold is configurable, and we use 200MB for this paper. Also, we profile target applications to predict idle time (e.g., when a program waits for a network response) and actively flush the log out from the buffer.

D. C^{2}SR *Resource Reconstructor*

C^{2}SR recovers *states of resources and proper content for each state* by inferring them from the recorded execution trace. It creates reconstructed resources where each of them consists of an automaton from the reconstructed states and reconstructed values for each state.

Resource Contents Reconstruction. C^{2}SR uses recorded syscalls to reconstruct resources. The states and contents of a resource during recording are inferred by analyzing how syscalls accessed the resource. For instance, `read()` syscall on a file (with the file pointer at the beginning of the file) that returned 'P' indicates that the file content should start with 'P'. As different resources may have different internal states and characteristics, we categorize syscalls that *access external resources* into three different types, as shown in Table II. We use different approaches for each type. Note that C^{2}SR focuses on syscalls that access external resources. Syscalls for internal resources (e.g., shared memory and signals) are not traced and reconstructed. They will be directly executed during the reconstructed execution. As the goal of C^{2}SR is reproducing an *attack delivery process* instead of faithfully replaying a particular vulnerability exploitation, internal resources are not our focus (Details in § VI). The complete list of our categorization, including justifications, can be found in Appendix § A.

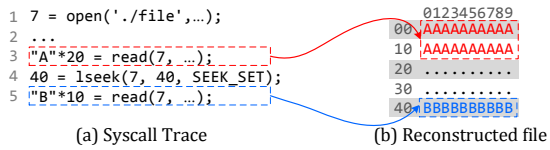
TABLE II. RESOURCE TYPES AND EXAMPLES.

Resource Type	Examples	Syscalls on the Resource
Random-Access	Files and Folder	`read()`, `readdir()`, `readlink()`, ...
Sequential-Access	Sockets, Pipe, Std. I/O	`send()`, `recv()`, `pipe()`, `read()`, ...
Timing-Dependent	Clock, Random Devices	`clock_gettime()`, `getrandom()` ...

1) Random-Access: Resources that permit random accesses to their contents belong to this category.

• *Reconstruction:* To access arbitrary contents of random-access resources, there are syscalls (e.g., `lseek()`) that can

specify the current access position (e.g., file offset) of the resource contents. In addition, when a resource is accessed, the current access position is automatically advanced by the number of bytes successfully accessed (i.e., read and written). Hence, we track syscalls that the current position to reconstruct states of a resource and contents associated with the states.



(a) Syscall Trace (b) Reconstructed file

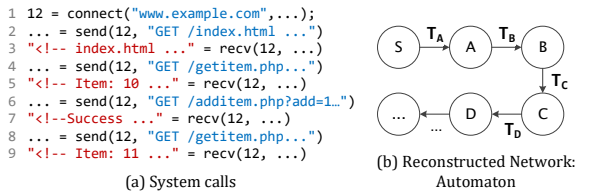
Fig. 6. Reconstruction of random-access resource.

– *Example:* Fig. 6-(a) and (b) show syscalls during recording and the reconstructed file respectively. It first opens a file (file handle is 7, returned at line 1) and reads 20 bytes of 'A' (Line 3). Then, it changes the file offset to 40 (Line 4) and reads 10 bytes of 'B' (Line 5).

C^{2}SR reconstructs the file content from the recorded `read()` and `lseek()`. Specifically, the first `read()` (Line 3) indicates that there are 20 bytes of 'A' from the beginning of the file. Then, `read()` (Line 5) happens after the `lseek()` (Line 4), which moves the file offset to 40, meaning that there are 10 bytes of 'B' in the file from the offset 40 (i.e., the current file position). Note that other parts (e.g., content between the offsets 20 and 39) are unknown as they were not accessed during recording. We use '.' to represent unknown (i.e., undefined hence unreproducible) content. If a reconstructed execution attempts to access the undefined content, an exception is raised as it indicates that the reconstruction is failed.

2) Sequential-Access: This category includes resources that can only be accessed sequentially. Like the random-access resources, a sequential-access resource may have internal states that determine the outcome of accesses to it (i.e., return values of syscalls on the resource), where the internal states may change each time it is accessed. The state changes are often done implicitly without explicit syscall invocations.

• *Reconstruction:* The internal state of a sequential resource is determined by its access history of the resource (i.e., a trace of syscalls on the resource). Hence, for each resource, C^{2}SR captures syscalls on the resource to reconstruct resource states and contents. Note that C^{2}SR assumes conservatively that any syscalls may change the internal state.



(a) System calls (b) Reconstructed Network: Automaton

State	Contents
Ⓐ	<!-- index.html ...
Ⓑ	<!-- Item: 10 ...
Ⓒ	<!-- Success ...
Ⓓ	<!-- Item: 11 ...

(c) Resource State Table

Transition	Condition
T_A	send("GET /index.html ...")
T_B	send("GET /getitem.php ...")
T_C	send("GET /additem.php?add=1 ...")
T_D	send("GET /getitem.php ...")

(d) State Transition Condition Table

Fig. 7. Reconstruction of sequential-access resource.

– *Example:* Fig. 7-(a) presents a syscall trace of a recorded execution that connects a website, *www.example.com*. There are 4 requests (Lines 2, 4, 6, and 8) and 4 corresponding responses (Lines 3, 5, 7, and 9 respectively). Note that each response is *dependent on itself and previous requests*. For instance, the response at line 3 depends on the request at line 2. Moreover, while the requests at lines 5 and 9 are identical, the responses are different because the server's *internal state was changed by the request at line 6* which added a new item. To this end, we devise an automaton as shown in Fig. 7-(b). Every request is a transition condition (Fig. 7-(d)), leading to a new resource state (Fig. 7-(c)). It also shows content of the resource on each state.

3) Timing-Dependent: If an access result (i.e., syscall's return) to a resource is dependent on *the time of the access*, it is a timing-dependent resource. For instance, `clock_gettime()` returns the current time and `read()` on a random device (e.g., "`/dev/random`") returns different values on every access, both depending on the time of invocation.

• *Reconstruction:* As the contents of timing-dependent resources do not depend on their internal states (i.e., access history), the reconstruction approaches for random-access and sequential-access resources are not applicable. In existing record-and-replay approaches, timing-dependent resource accesses are faithfully replayed with its strict order. However, syscalls on timing-dependent resources can be different across runs because C^{2}SR allows user-interactions and partial execution reconstruction. In other words, a reconstructed execution may have additional/missing syscalls, making it challenging to apply the same method as the existing techniques do.

– Our Approach: Timeline Reconstruction. C^{2}SR reconstructs contents of timing-dependent resources by projecting timing-dependent resource accesses into a timeline that is reconstructed from the recorded execution's timing information (i.e., timestamps of syscalls). The intuition of timeline reconstruction is that we record timestamps of all syscalls to abstract the ideal timings of the syscalls, including the timing-dependent syscalls, which we call "timeline." Then, we project the ideal timeline by fitting (e.g., shrinking/stretching) the timeline into the reconstructed execution context. Specifically, it infers two timelines, a timeline for the recorded execution and another timeline for the reconstructed execution. Then, it projects the recorded timeline to the reconstructed timeline. C^{2}SR leverages the projection to identify proper values for syscalls on timing-dependent resources.

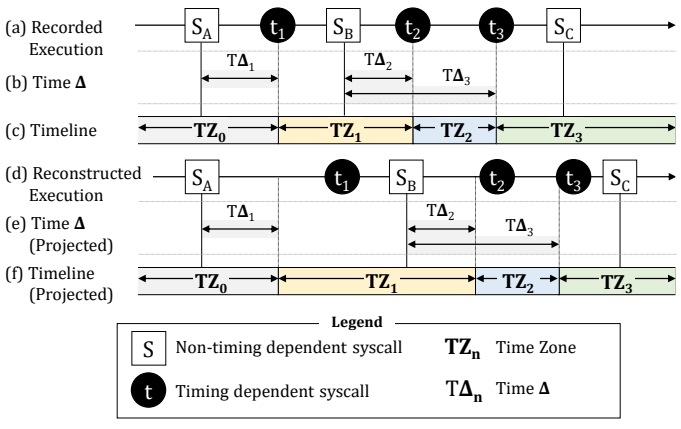


Fig. 8. Example of Timeline Reconstruction.

– ***Concepts:*** We introduce three concepts, *Time* Δ, *Time Zone*, and *Timeline*, to explain how we project a timeline from a recorded execution to a reconstructed execution.

First, TimeΔ (or TΔ) is a distance between a timing-dependent syscall and its immediately previous syscall. For example, in Fig. 8-(a), TΔ_1 is computed by subtracting timestamps between t_1 and S_A (the syscall right before t_1).

Second, a time zone is a timespan between two consecutive timing dependent syscalls. In Fig. 8-(c), time zones are annotated inside the timeline as TZ_n. Given two consecutive timing dependent syscalls t_n and t_{n+1}, a time zone TZ_n represents the timespan between t_n and t_{n+1}. For instance, in Fig. 8-(c), TZ_1 is the timespan between the two timing dependent syscalls t_1 and t_2.

Third, a timeline is a sequence of time zones, obtained from an execution trace by analyzing timing dependent syscalls. Intuitively, a timeline (and time zones) provides a guideline for which time zone a syscall belongs. For example, in Fig. 8-(c), for t_1, the closest time zone is TZ_1, if we use the starting of the timezone to attribute. Similarly, the closest time zones for t_2 and t_3 are TZ_2 and TZ_3, respectively.

– ***Timeline Projection:*** Given TΔ values and a timeline computed from a recorded execution, we project the timeline into a reconstructed execution. We first identify non-timing dependent syscalls. Recall that each TΔ_i is calculated by subtracting timestamps of t_i and the t_i's immediately previous syscall. During projection, we find corresponding non-timing dependent syscalls. For each of them, we apply the TΔ_i computed from the recorded execution, to determine when a time zone TZ_i should start.

– *Example:* Fig. 8-(f) shows an example. The beginning of TZ_1 is computed by adding the timestamp of S_A and $T\Delta_1$. Similarly, TZ_2 and TZ_3 are obtained by computing the timestamp of S_B + $T\Delta_2$ and $T\Delta_3$, respectively.

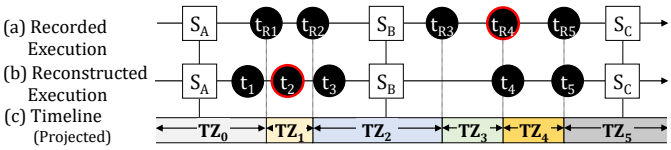
Fig. 8-(f), the projected timeline provides a correct guidance for the reconstructed execution. Observe that the syscalls between Fig. 8-(a) and Fig. 8-(d) happen in a different speed. In particular, syscalls in the reconstructed execution happen slower than the recorded execution. With the timeline projection, for all t_1, t_2, and t_3 in the reconstructed execution, their closest projected time zones are TZ_1, TZ_2, and TZ_3, respectively. This essentially means that we correctly provide the same recorded values for t_1, t_2, and t_3.

Summary. C^{2}SR reconstructs contents of resources according to the logical structure of resources (i.e., how the resources should be accessed and what values should be expected). We analyze all existing Linux/Unix syscalls and categorized their target resources into three different types: random-access, sequential-access, and timing-dependent. C^{2}SR aims to reproduce consistent resource accesses with respect to the accesses during recording. C^{2}SR detects failures in reconstructed executions by monitoring accesses to undefined resources, content, and early execution termination (e.g., caused by software faults).

E. C^{2}SR *Execution Reconstructor*

C^{2}SR's runtime modules intercept syscall invocations during the reconstructed execution. When the execution tries to access resources that were reconstructed, C^{2}SR emulates the resource accesses using the reconstructed resource states, contents, and timeline. However, it is possible to have a wrong timeline projection. Specifically, with the current method, if there are additional syscalls or syscalls in a reconstructed execution happen faster than its recorded execution, our timeline projection may fail due to additional syscalls (caused by user-interactions and non-determinism). Hence, we propose *Timeline Projection Adjustment (TPA)* to handle such failures.

Timeline Projection Adjustment (TPA). When the timeline projection is failed (leading to a failure in execution reconstruction), C^{2}SR tries to adjust the timeline projection. It aims to find a new timeline with the adjustment that can lead to a successful reconstruction. Specifically, for each syscall on a timing-dependent resource in a trace, C^{2}SR tries to find another appropriate time zone. Note that searching for all appropriate timelines can be practically infeasible due to the large searching space (e.g., with n time zones and m syscalls on timing resources, there exist m^n projections). Hence, we start finding alternative timeline projections from the time zones that are close to the execution failure (i.e., where the execution failed). Then, we search the alternatives in a backward direction.



t_1	t_2	t_3	t_4	t_5	TZ_1	TZ_2	TZ_3	TZ_4	TZ_5
3.355	3.825	4.325	6.435	7.145	3.58	4.13	5.66	6.38	7.1

(d) Timestamps of timing dependent syscalls (t_1~t_5) and Time Zones (TZ_1~TZ_5)
(Values in milliseconds)

	t_1	t_2	t_3	t_4	t_5
TZ_1	0.225	0.245	0.745	2.855	3.565
TZ_2	0.775	0.305	0.195	2.305	3.015
TZ_3	2.305	1.835	1.335	0.775	1.485
TZ_4	3.025	2.555	2.055	0.055	0.765
TZ_5	3.745	3.275	2.775	0.655	0.045

(e) Distances between Time Zones and Syscalls on Timing Dependent Resources
(Darker background indicates larger distance values)

	t_1	t_2	t_3	t_4	t_5
Ground Truth	TZ_1 (3.58)	N/A (Any)	TZ_2 (4.13)	TZ_3 (5.66)	TZ_5 (7.1)
First Attempt	TZ_1 (3.58)	TZ_1 (3.58)	TZ_2 (4.13)	TZ_4 (6.38)	TZ_5 (7.1)
Second Attempt	TZ_1 (3.58)	TZ_1 (3.58)	TZ_2 (4.13)	TZ_5 (7.1)	TZ_5 (7.1)
Third Attempt	TZ_1 (3.58)	TZ_1 (3.58)	TZ_2 (4.13)	TZ_4 (6.38)	TZ_4 (6.38)
Forth Attempt	TZ_1 (3.58)	TZ_1 (3.58)	TZ_2 (4.13)	TZ_3 (5.66)	TZ_5 (7.1)

(f) Timeline Projections and Adjustments
(Red background indicates the root cause of the failure)

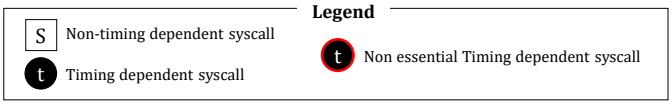


Fig. 9. Running example of Timeline Projection Adjustment (TPA).

We use an example of reconstructing an HTTPS webpage loading in Firefox to show how the timeline projection adjustment (TPA) works. Fig. 9-(a) and Fig. 9-(b) show syscalls

from a recorded execution and its reconstructed execution respectively. $t_{R1\sim R5}$ and $t_{1\sim 5}$ are syscalls on timing-dependent resources (i.e., `time()`), and $S_{A\sim C}$ are syscalls on non-timing-dependent resources. To facilitate the discussion, we assume that all t_x are `time()` and all S_x are `send()` where x is an integer. The trace is generated by an execution that establishes an SSL connection. In the recorded execution, t_{R1}, t_{R2}, t_{R3}, and t_{R5} are providing seed values for the SSL session key creation. In the reconstructed execution, t_1, t_3, t_4, and t_5 are the corresponding syscalls for seed values. Providing correct seed values is important, otherwise it will create a wrong SSL session key, causing a failed execution reconstruction. The wrong SSL key typically causes an early exit of the reconstructed execution. t with a red border (i.e., t_{R4} and t_2) is a syscall that is not critical to the execution (i.e., not relevant to the SSL keys), meaning that not providing a correct value for it does not cause an execution failure. In our case, it is a call from JavaScript library to measure its performance.

This example includes two scenarios: the reconstructed execution has an additional syscall and a missing syscall compared to the recorded execution. First, observe that between S_A and S_B, the recorded execution has two syscalls while the reconstructed execution has three. There is an additional syscall t_2. Second, between S_B and S_C, the recorded execution has three syscalls while the reconstructed execution has two, missing a syscall t_{R4}. Note that those additional syscalls are non-essential. However, due to those differences, it is challenging to find corresponding syscalls between the recorded and reconstructed executions. Fig. 8-(d) shows timestamps of syscalls on timing-dependent resources in the reconstructed execution and the projected time zones (Fig. 8-(c)). Fig. 8-(e) presents distances between time zones and syscalls that will be used to find alternative timeline projections.

– *Timeline Projection Adjustment Example:* Fig. 8-(f) shows a table that summarizes how C^{2}SR finds a working alternative timeline projection, when a reconstructed execution fails. Specifically, the first row shows the ground-truth, meaning that with the assignments (i.e., t_1=TZ_1, t_2= any values, t_3=TZ_2, t_4=TZ_3, and t_5=TZ_5), the reconstructed execution will succeed. However, the first timeline projection has a different timeline projection from the ground-truth. Specifically, the first timeline assigns time zones that are closest to the syscalls. Observe that TZ_4 is the closest time zone to t_4 as shown in Fig. 8-(e), while the correct time zone for t_4 is TZ_3. In the following paragraphs, we show how C^{2}SR automatically finds the correct timeline (i.e., the correct time zone assignments).

1) First Attempt: For the initial assignment, t_1, t_2, t_3, t_4, and t_5 are assigned to the time zone TZ_1, TZ_1, TZ_2, TZ_4, and TZ_5 respectively as these time zones are the closest ones to each of the syscalls. It fails at S_C as TZ_4 is assigned to t_4 while the desired time zone for t_4 is TZ_3.

2) Second Attempt: C^{2}SR looks for a different timeline from the failure in a reverse direction. Specifically, we first look at syscalls after the last successful syscall, S_B. Given the t_4 and t_5, we essentially try the second smallest combination of distances from Fig. 8-(e). As the assignments of t_4 = TZ_5 and t_5 = TZ_5 results in the second smallest distance (0.7 = 0.655+0.045), we try the new timeline. The second trial fails again.

3) Third Attempt: We find another time zone assignments between t_4 and t_5. There are two assignments that lead to the third smallest distance. First, (t_4 = TZ_4, t_5 = TZ_4) results in 0.82 (= 0.055+0.765) and (t_4 = TZ_3, t_5 = TZ_5) leads to 0.82 (= 0.775+0.045). We first try t_4 = TZ_4 and t_5 = TZ_4. However, it fails too.

4) Fourth Attempt: We try the other assignment: t_4 = TZ_3 and t_5 = TZ_5. This trial is successful.

Summary. Timeline Projection Adjustment (TPA) handles failed reconstructed executions due to incorrect timeline projections. It systematically searches alternative timelines that can lead to a successful reconstruction based on the distances between projected time zones and syscalls on timing dependent resources.

V. Evaluation

Our prototype of C^{2}SR, including recorder and resource reconstructor, is written in C++ (11,045 SLOC). We leverage the `LD_PRELOAD` environment variable to intercept library calls that invoke syscalls. Our resource reconstructor is also written in C++ (2,871 SLOC). Experiments are on a machine with Intel i7-4770 3.4GHz CPU (4 cores), 16GB RAM, 512GB SSD, and LinuxMint 19.2 (64-bit).

Program Selection. We use a total of 26 diverse programs to evaluate different aspects of C^{2}SR. For each experiment, we choose a different set of programs. We explain how and why we choose the programs as follows.

1) For Runtime Overhead of Recorder (§ V-A): We use SPEC CPU2017 Integer (10 programs), 5 web browsers (Firefox, Arora, Midori, Qupzilla, and Opera) for running two JS/DOM benchmarks (Octane [78] and Speedometer [90]), and three web servers (Apache, Nginx, Lighttpd) to measure the runtime performance of serving HTTP(s) requests. We choose SPEC CPU2017 as it is the standard performance evaluation suite, while we are aware of that they do not invoke syscalls extensively, resulting in favorable (and possibly misleading) evaluation results. We omit SPEC CPU2017 Floating Point benchmarks due to the same reason as they have almost no syscalls. Instead, we run more realistic workloads: JS/DOM benchmarks on web browsers and 3 popular web servers that represent server-side applications. Moreover, we use 8 client applications including web browsers, email client, and messengers to measure the performance during the initialization of the processes (when programs invoke syscall intensively).

2) For Space Overhead of Recorder (§ V-B): To understand space overhead of C^{2}SR in realistic settings, we choose 6 representative programs that are popular and commonly targeted by advanced cyberattackers (e.g., common targets for phishing). It includes a web browser (Firefox), an email client (Thunderbirds), instant messengers (Skype and Yakyak), and web servers (Apache and nginx). We did not use SPEC CPU2017 as they do not invoke syscalls extensively, leading to a low space overhead. Other programs (e.g., other browsers and FTP/IRC clients and servers) are not selected as they incur less space overhead than the chosen programs.

3) For Efficiency of Partial Execution Reconstruction (§ V-C): We use 5 web browsers and 5 server programs (3

web servers, 1 IRC server, and 1 FTP server). We select web browsers as they are long-running programs and support concurrent autonomous tasks (e.g., browser tabs) that C^{2}SR primarily targets. Server programs are long-running programs where each request is independent from others.

Runtime Overhead of the Post-processing. In § III, C^{2}SR has an offline post-processing phase. Note that it merely slices the traces based on resources, and it is not a particularly expensive process, with costs growing linearly with the trace size. It is a one-time effort for each recording. During our evaluation, no post-processing tasks take more than a minute.

A. Runtime Overhead

SPEC CPU2017. Fig. 10 shows normalized runtime overhead on SPEC CPU2017 Integer programs for recording and execution reconstruction with the reference inputs. The first 10 bar graphs present recording overhead while the last 10 bar graphs show the overhead of reconstructed executions. C^{2}SR incurs negligible runtime overhead (less than 3% for all cases and 0.8% on average) for recording. For reconstructed executions, C^{2}SR incurs only 2% runtime overhead on average, which is slightly slower than recording as C^{2}SR needs to redirect syscalls to the reconstructed resources (§ IV-D). In addition, as SPEC CPU2017 programs are mostly deterministic, we do not observe any instances that C^{2}SR uses the timeline projection adjustment algorithm, and every reconstructed execution successfully reproduced the target execution in their first attempts.

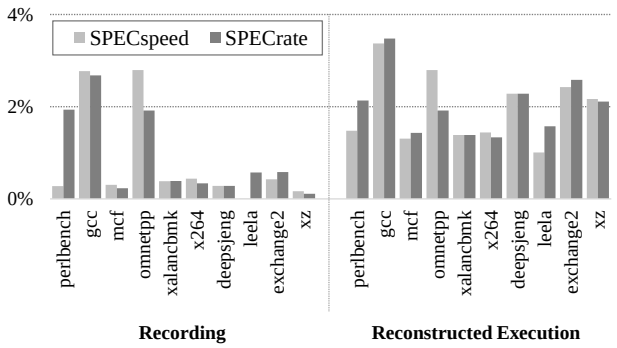


Fig. 10. Runtime overhead on SPEC CPU2017.

Web Browsers and Web Servers. We evaluate the recording performance of C^{2}SR on web-browsers using JS/DOM benchmarks as shown in Table III. Specifically, we use Octane [78] and Speedometer [90] to measure the performance of each browser application with and without C^{2}SR's recorder. The numbers are normalized and represent overhead with respect to the performance without C^{2}SR. In most cases, C^{2}SR slows down the execution by less than 2%, except the Opera in Speedometer case which results in 3.84% slow down. The results confirm that C^{2}SR is highly practical, causing low overhead. We also evaluate C^{2}SR's recording performance on three popular web-servers leveraging the apache benchmark program [1]. Specifically, we use the tool to generate 1,000,000 requests with 8 threads and then measure the elapsed time on each web server with and without C^{2}SR's recorder. As shown in Table IV, the recoding overhead is negligible. Note that processing a trace to create virtualized resources is an offline process hence does not affect the runtime performance. In addition, our buffering optimization (Details in § IV-C) contributes to the low performance overhead.

TABLE III. RUNTIME OVERHEAD ON JS/DOM BENCHMARKS.

Benchmark	Firefox	Arora	Midori	Qupzilla	Opera
Octane [78]	1.93%	1.31%	1.62%	0.8%	0.9%
Speedometer [90]	0.5%	0.5%	1.83%	1.52%	3.84%

TABLE IV. RUNTIME OVERHEAD ON WEB-SERVER BENCHMARKS.

	Apache	Nginx	Lighttpd
Overhead (Normalized)	2.7%	3.2%	3.1%

Syscall-intensive Client Applications. As SPEC CPU2017 and web browsers/clients benchmarks are not syscall intensive, we choose a few client applications to measure the overhead of C^{2}SR. In particular, we measure runtime overhead while the applications are initializing, as initialization phases of applications are often syscall intensive. We pick 5 web browsers (Firefox, Midori, Arora, Qupzilla, and Lynx) and Thunderbirds, Skype, and HexChat.

For all applications, as they have the graphical user interface (GUI), we consider the initialization is complete when the first GUI component is created. For each program, we run the application 10 times to measure overhead during recording and reconstruction, and then we take the average.

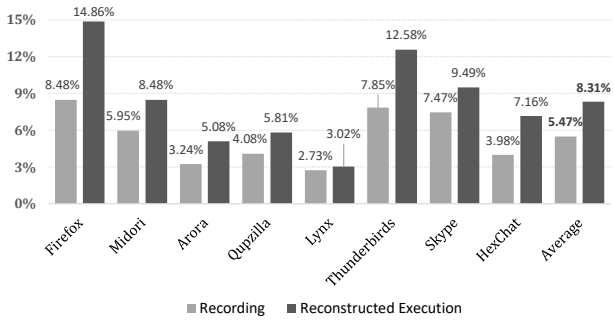


Fig. 11. Runtime overhead on Client Applications during Initialization.

Fig. 11 shows the result. The average is 5.47% for recording and 8.31% for reconstruction. Large applications such as Firefox (8.48% and 14.86%), Thunderbirds (7.85% and 12.58%), and Skype (7.47% and 9.49%) incur more overhead than others as they issue more syscalls during initialization (e.g., loading configuration files and connecting servers to load user profiles). We believe the overhead is reasonable as the frequency of syscall invocations becomes lower after the initialization, meaning that in practice, the overhead of C^{2}SR is lower than the overhead presented in Fig. 11.

B. Space Overhead

We measure the space overhead of execution traces generated by C^{2}SR. Note that the traces include all inputs and outputs to/from a program, meaning that the traces always take more space than all the inputs and outputs.

SPEC CPU2017. Most SPEC CPU programs have multiple reference inputs. We use all of them and add all the logs to measure the space overhead as shown in Table V. The second column shows the total accumulated trace size for each program. The next three columns show the size of non I/O syscalls, input related syscalls, and output related syscalls in the trace. Note that the size of I/O (input and output)

TABLE V. SPACE OVERHEAD ON SPEC CPU2017.

Program	Raw (Total)	Non I/O	Input	Output
perlbench	57.7 MB	1,009 KB	50.3 MB	6.4 MB
gcc	218.3 MB	11.8 MB	92.4 MB	114.1 MB
mcf	2.2 MB	85 B	2.2 MB	2.5 KB
omnetpp	56.8 KB	9.9 KB	46.8 KB	104 B
xalancbmk	119.1 MB	3.9 MB	55.4 MB	59.7 MB
x264	379.8 MB	112 KB	375.1 MB	4.6 MB
deepsjeng	45.9 KB	40 B	1,058 B	44.8 KB
leela	1.5 MB	451 B	51.3 KB	1.45 MB
exchange2	15.4 KB	888 B	2.9 KB	11.6 KB
xz	16.97 MB	99 B	16.9 MB	3.1 KB
Total	795.6 MB	16.9 MB	592.3 MB	186.4 MB
Zip	154.9 MB	3.1 MB	118.9 MB	32.9 MB

related logs are almost identical to the I/O contents (e.g., input and output file sizes). Non I/O column shows a space overhead without the I/O contents. From the total, Non I/O syscalls only takes 2.12% of the total trace size. This shows that C^{2}SR does not cause much additional space overhead beyond logging input and output. Input is required to drive reconstructed execution and output is needed for verifying the reconstructed execution is successful.

Fig. 12 presents more detailed analysis of the space overhead caused by C^{2}SR. It illustrates the percentage of input, output, and non-I/O syscalls in execution traces. The y-axis represents each program and the x-axis shows the percentage. Observe that all the programs except for `omnetpp`, more than 90% of the trace is occupied by input and output. For `omnetpp`, non-I/O syscalls take 17.4% because the total size of the trace is small (56.8 KB, as shown in Table V), making the proportion of the non-I/O syscalls significant.

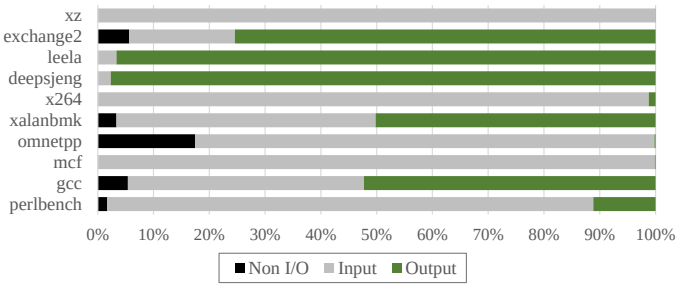


Fig. 12. SPEC CPU2017 Space Overhead Breakdown.

TABLE VI. SPACE OVERHEAD ON THE CLIENT PROGRAMS.

	Firefox	Thunderbirds	Skype	Yakyak	Apache	nginx
Raw	0.9 GB	126.1 MB	497.2 MB	314.4 MB	433.3 MB	289.9 MB
Zip	39.8 MB	8.1 MB	36.8 MB	20.2 MB	23.8 MB	16.7 MB

Representative Client Programs. SPEC CPU2017 programs are computation-intensive, but not syscall intensive. As C^{2}SR's space overhead is dependent on the number of syscall invocations, we run an additional experiment with representative client programs. We run 5 heavy websites (Facebook, Twitter, CNN, Gmail, and New York Times) for 20 minutes on Firefox (e.g., browsing 37 webpages, reading 7 news articles, searching 10 keywords on Google), reading and writing emails for 20 minutes in Thunderbirds, accessing HTML/PHP files on web servers powered by Apache and Nginx, and using Skype and yakyak (for Google Hangout) to send/receive text messages/files for 20 minutes. As there is no standard workload available for them, we manually use the applications actively for the given amount of time. The results are shown in Table VI. The first row shows the trace sizes and the second row represents compressed (via zlib [105]) trace sizes. While the size of traces is non-trivial, considering that all input/output contents (e.g., contents of network servers) must be stored for post-mortem forensic analysis, we argue that our space overhead is acceptable for our purpose. Note that the compression significantly reduces the log size. We observe that the compression works better for real-world applications because, in part, they have more repetitive trace patterns.

C. Partial Execution Reconstruction

To evaluate the effectiveness of C^{2}SR's partial execution reconstruction, as shown in Table VII, we use 5 server (top 5 rows) and 5 client programs (bottom 5 rows). For each program, we record an execution and then reconstruct an execution of a single (randomly picked) request. Specifically, we pick 10 random requests (e.g., a single web-page view in a web browser or a single session in a server program) and present a median value in Table VII. Note that the number of syscalls that need to be reconstructed to successfully reproduce a single request is very small (i.e., less than 1% of the entire syscalls). As C^{2}SR does not need to run a target program from the beginning, it is highly efficient and effective in reproducing partial executions from a long-running application for post-mortem analysis. Also, the fourth column shows the number of syscalls that access timing-dependent resources. In web-browsers, there are often more timing-dependent related syscalls as those are often used in GUI-related operations (e.g., user interaction, animations). In server programs, those are often used for performance profiling and logging functionalities. The last two columns (i.e., Recon. and Syscall) show the number of reattempted execution reconstructions and the number of syscalls (on timing-dependent resources) that were reassigned alternative timezones. In general, client programs (e.g., web-browsers) require more retrials (e.g., 5 ~ 14) while server programs require at most 3 retrials. Even in the worst case, Firefox, it only retries 14 times. Note that the numbers in the sixth columns are the ones that C^{2}SR's timeline projection adjustment algorithm searched for. Our manual inspection reveals that the retrials are caused by the frequent use of timing-dependent resources for en/decryption and third-party JS libraries.

TABLE VII. RECONSTRUCTION OF LONG-RUNNING PROGRAMS.

Program	# of syscalls			# of retrials	
	Total	Reconstructed	Timing-dep.$^\alpha$	Recon.$^\beta$	Syscalls$^\gamma$
Firefox [31]*	482.6M	381.5K (**0.07%**)	32.5M (6.7%)	14	21
Midori [12]*	261.7M	351.4K (**0.13%**)	59.6M (22.7%)	6	10
arora [9]*	75.6M	68.3K (**0.9%**)	7.9M (10.4%)	9	14
qupzilla [85]*	51.9M	45.5K (**0.08%**)	13.6M (26.2%)	6	8
lynx [61]*	5.5M	2.3K (**0.04%**)	480.2K (8.6%)	5	7
Apache [8]**	6.9M	141 (**0.002%**)	620M (8.9%)	3	4
nginx [76]**	5.5M	107 (**0.001%**)	568K (10.1%)	2	4
lighttpd [58]**	4.1M	81 (**0.001%**)	206K (4.92%)	2	2
InspIRCd [40]**	1.6M	2.3K (**0.04%**)	8.3K (0.5%)	0	0
proftpd [83]**	3.3M	2.3K (**0.04%**)	10K (0.29%)	0	0

*: Client programs, **: Server programs, α: Syscalls on timing-dependent resources, β: # of execution reconstruction retrials, γ: # of syscalls with *alternative* time zone assign.

D. Case Study

We present two case studies to show how C^{2}SR can effectively reconstruct attacks and allow forensic analysts to use program analysis tools to facilitate the analysis.

Case 1: Investigation of a Fileless JavaScript Attack. We use a fileless JavaScript attack to show how C^{2}SR can reconstruct an interactable partial execution for forensic analysts. The attack is created based on an existing literature [88]. We only use a single browser tab, in this case, to focus on demonstrating the effectiveness of interactable reconstructed execution.

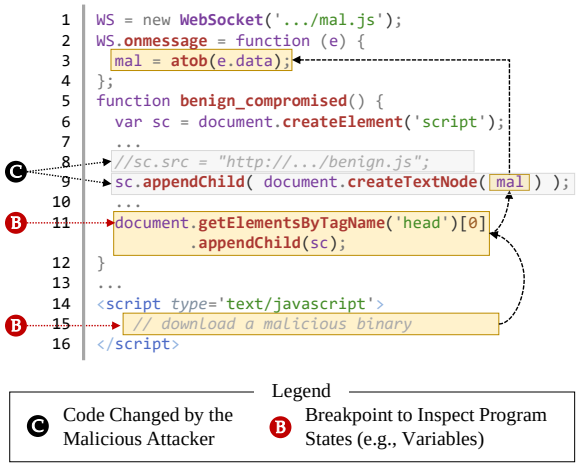


Fig. 13. Investigating a Fileless Attack via JavaScript.

1) Experimental Setup: The victim uses Firefox to navigate a web page that contains a malicious JavaScript file, which fetches another malicious payload that eventually downloads malware. From the malicious file downloading, a forensic analyst tries to trace back to the origin of the attack and how the code that downloads the malware binary was delivered.

2) Investigation: With the reconstructed execution by C^{2}SR, the forensic analyst identifies that malware is downloaded from a web page, including a JavaScript code snippet shown in Fig. 13. Specifically, the script code block (Lines 14-16) is added dynamically to download malware. Using *DevTools* [30], the analyst sets a breakpoint at line 15 (B). She realizes that the script code block is dynamically added by comparing the HTML file content delivered through the network. She then restarts the execution and uses *DevTools* to set breakpoints on DOM element change events in order to identify how the code block is dynamically added. The breakpoint stops at line 11, showing the malicious code. She further traces back to the origin. The script code was injected at line 9. Note that the variables (e.g., `mal` at line 9) and DOM elements (e.g., `sc` at line 11) related to the malicious payload delivery are only available at runtime, requiring the analyst to use *DevTools* for inspection. The user interactions required to set breakpoints and inspect the variables incur additional instructions and syscalls. C^{2}SR tolerates them as additional syscalls do not access completely unobserved external resources, hence consistently reproducible.

To this end, she realized that the HTML file is compromised that line 8 (C) was disabled and line 9 was added by the attack to inject the DOM element containing malicious code. Finally, she follows the `mal` variable which contains malicious code, eventually discovering that it receives the malicious code via WebSocket at lines 1-4.

Case 2: Attack Delivery Analysis in a Channel (Single Autonomous Task) of HexChat. `HexChat` is an IRC client program that supports multiple channels (i.e., chatting sessions) concurrently. We choose this program to demonstrate an example of an application that runs multiple autonomous tasks concurrently besides previous web browser cases. The program has a directory traversal vulnerability [23] that can be exploited to modify arbitrary files on the client's file system.

1) Experimental Setup: The victim uses `HexChat` and opens 10 channels for 2 days. Even though the user did not actively chat, the program constantly received messages from other users and displays them. For the 2 days, it received 63,738 messages from 10 servers, and it receives a malicious message from the channel 10, and the message exploited the vulnerability to corrupt the apache web server's configuration file. The corrupted configuration file may break the web server or make it use the insecure default configuration.

2) Result and Analysis: C^{2}SR reconstructs *a part of the single exploited channel 10* (B) as shown in Fig. 14. From the entire trace, we identify the malicious `write()` syscall that corrupts the apache configuration file (i.e., *apache2.conf*). From the malicious file write, we trace back the socket that received the malicious payload and the channel that the socket belongs to, which is the channel 10. To this end, we use the first `getaddrinfo()` API of the channel as the beginning of partial execution (S_{BEGIN}).

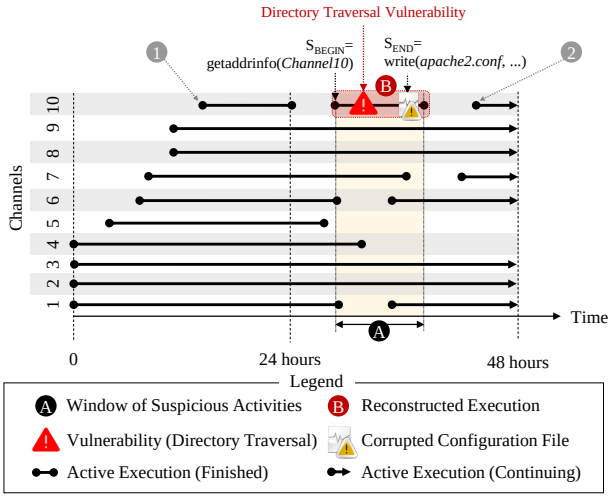


Fig. 14. Reconstructed partial execution of `HexChat`.

C^{2}SR starts an execution from with the collected trace. Once the execution starts, we open a new connection window and type the channel name. Note that in the channel 10, there are three executions (1, B, and 2) that connect to the same server. C^{2}SR asks the investigator to choose the execution from them. If the analyst mistakenly chooses 1 or 2, S_{END} (the malicious syscall) cannot be reached, and thus the reconstruction will fail. If the second execution (B) is chosen, C^{2}SR will reproduce the entire attack delivery process including the exploitation process.

To analyze the incident, we use Taintgrind [48] (a taint analysis plug-in for Valgrind [74]) on the reconstructed execution. It supports reverse taint analysis (rtaint [24]), that trace back to the origin of a certain value. In this example, we first trace back to the file handler used in the malicious `write()` (line 17), and then identify how the filename was composed

and used to create the file. Fig. 15 shows how the reverse taint analysis identifies the source of the message and its content, from the suspicious filename at line 15.

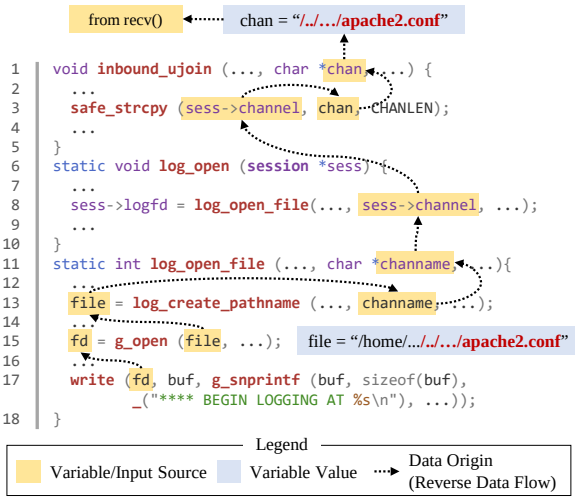


Fig. 15. Reconstructed partial execution of `HexChat`.

Note that there exist record-and-replay techniques that allow to integrate with existing program analysis infrastructures such as Pin [60] and gdb [34]. Such tools can analyze replayed executions. However, they require the existing program analysis tools to interact with the replayed execution through the replayer. As a result, limited program analysis infrastructures and plug-ins are often supported and analysis results may need to be adjusted as the tools are not directly applied. Unlike them, C^{2}SR allows existing program analysis techniques to directly interact with the reconstructed execution. Hence, diverse analysis tools such as Taintgrind with rtaint [24] can be directly used. In addition, C^{2}SR can reconstruct only a small part of the channel 10 execution without reconstructing 80% of the messages that are irrelevant to the attack.

VI. Discussion

Reconstructing Non-deterministic Executions. C^{2}SR is not a deterministic replay technique and it might not be able to reconstruct a highly non-deterministic execution at the first attempt. To mitigate this issue, we propose an algorithm, called Timeline Projection Adjustment (see § IV-E), to systemically search for the correct order of non-deterministic events. We demonstrate that it can successfully reconstruct the complete execution of the attack delivery process. Highly non-deterministic executions, such as concurrency bugs that require to reproduce the exact execution states of each thread (e.g., replay each instruction in the same order of the recording) are out of the scope. Instead, we focus on reproducing a *deterministic attack delivery process*. However, while C^{2}SR does not guarantee to reproduce the non-deterministic behaviors, they may occur in the reconstructed execution, if the behaviors are not rare. In practice, a forensic analyst may try to reconstruct a non-deterministic execution multiple times to successfully reproduce the execution.

Inconsistent Resource Access. User interactions for inspections may cause new resource accesses that were not observed during recording. For instance, users can directly execute JavaScript code snippets on the *DevTools*'s console to access new external resources. We handle such cases as follows. If it is an available resource in the reconstruction time, (e.g., reading a local configuration file of an analysis tool), we allow to do and resume the reconstruction process. If the resource is not available (e.g., accessing external resources such as loading a webpage), we consider it as inconsistent resource access, leading to a reconstruction failure.

Attacks against C^{2}SR. It is possible that an attacker can purposefully create malicious payloads (e.g., JavaScript programs) that make the execution reconstruction by C^{2}SR difficult. In particular, since C^{2}SR runs a searching algorithm (Timeline Projection Adjustment algorithm in § IV-E) when it fails to provide correct values for timing-dependent syscalls (e.g., `time()`), attackers can craft payloads that make the algorithm run for a long time, hindering execution reconstruction and forensic analysis. For instance, assume that there are two syscalls: t_x for a timing dependent syscall (e.g., `time()`) and s_y for a non-timing dependent syscall (e.g., `read()`). If s_y uses the return value of t_x as one of the arguments, then the execution of s_y is dependent on t_x, meaning that reconstructing the correct value for t_x is required for successful execution of s_y. If an attacker creates a program that has many timing dependent syscalls between t_x and s_y, it causes a large searching space for our TPA (Timeline Projection Adjustment) algorithm (§ IV-E). For example, consider the original program contains "t_x, s_y". A possible code that can impose challenges to C^{2}SR is "t_x, t_1, t_2, t_3, ..., t_n, s_y", where n is the number of added timing-dependent syscalls. It can exploit the fact that C^{2}SR does not try to reconstruct values for timing-dependent resources faithfully. However, while it might delay the reconstruction significantly, this does not break our analysis and it is still possible to reconstruct the correct execution.

Log Integrity. In this paper, we assume that our event logger and logs generated are not compromised and tampered. We focus on algorithms and systems to reconstruct an exploit execution from the log. Protecting the integrity of the recorder and logs is an orthogonal problem that we can leverage existing solutions to mitigate [3], [4], [28], [86], [19], [63], [100], [89], [79], [46].

Generality. C^{2}SR is applicable to diverse applications. In particular, it can effectively reconstruct a partial execution when a target program execution is consisting of multiple autonomous tasks (i.e., tasks that are supposed to be independent of other tasks). For example, a document or window in multi-document/window applications (e.g., tab-based web browsers, messaging programs, and multi-tab text editors) is an autonomous task. Among them, web browsers are a popular target in cyberattacks. Hence, we focus on web browsers in this paper. However, our design is not limited to them.

Identifying a Task to Reconstruct. In this paper, we focus on the effective and efficient reconstruction of the cybercrime scene. Detecting a malicious task or identifying a cybercrime scene is out of the scope of this work. In practice, a forensic analyst can typically narrow down a part of execution (e.g., a tab executing malicious payload), while she spends a significant amount of time and effort to analyze the suspicious execution repeatedly for hypothesis testing. while pinpointing a malicious task can be difficult, it can be done by ruling out obviously benign network connections/system events.

Space Overhead. The current implementation of C^{2}SR uses *zlib* library [105] to compress execution traces. While the log size is reasonable, it can be further reduced by leveraging recent studies [99], [38], [64].

Debugging Capability. There are approaches [45], [37] that aim to provide a replay of a recorded execution for debugging purposes. While effective, [45] requires modifications to the target program. [37] aims to handle additional code inserted for debugging by developers. It also requires modifications to the target program. C^{2}SR differs from them because (1) it allows users to apply interactive debugging tools (e.g., *DevTools* [30]) and (2) it does not require modification of the target program. [13] also supports interactive debugging, while it only supports web browsers and requires modifications of the web browsers, as discussed in § II-D.

VII. Related Work

In addition to our earlier discussion (§ II-D), we discuss other related work in this section.

A. Record-and-replay Techniques

1) RR [77]: State-of-the-art Record-and-replay Technique: We compare C^{2}SR with RR [77], the state-of-the-art record-and-replay technique for debugging. In particular, we use the same set of benchmark programs used by RR with the similar configurations: cp, make, octane, htmltest, and sambatest. Note that the cp benchmark in [77] uses 15,200 files constituting 732MB of data. Since the files are not publicly available, we use 720MB of data consisting of 37,356 files.

Recording Overhead. The average recording overhead of C^{2}SR is 5.74% which is significantly lower than the RR, which incurs 2.54x overhead (according to [77]). This is because RR enforces various changes in the software and environment. For instance, it forces to use a single thread, causing significant overhead (7.85x) on the make benchmark which originally uses 8 threads. C^{2}SR does not have such restrictions, resulting in a near native speed in the recording.

Side-effects. To reduce the sources of non-determinism, RR implements various restrictions for executions during recording. First, it only runs one thread at a time. This slows down the recording execution as discussed above. More importantly, this will result in a significantly different execution at runtime. For instance, with RR's recorder, many concurrency bugs may disappear due to the restriction. C^{2}SR does not have such restriction. However, C^{2}SR is also ineffective in reproducing concurrency bugs. Second, RR interferes with the context switching as well, performing preemptive context switching. An execution under the RR's recorder may exhibit different context switches from the original execution without the RR's recorder. C^{2}SR does not interfere with the context switching. Third, RR uses `ptrace` to implement various hooks, while C^{2}SR hooks libraries which is faster than the `ptrace`. As reported in § V, C^{2}SR's recording overhead is much less than RR's recording overhead, allowing C^{2}SR's recorder to capture executions that are close to the original executions.

Robustness. C^{2}SR's recording capabilities are entirely implemented as a library while RR leverages more robust infrastructures such as `ptrace`. If program code is corrupted during recording and reconstruction, C^{2}SR may be affected (i.e., compromised and fail), while RR would be robust.

2) Other Record-and-replay Techniques: We also compare with other existing record-and-replay techniques [80], [5], [39], [70], [27], [87], [91], [21], [93], [52], [91], [53], [17], [67], [84], [28], [42], [43], [44], [71], [13], [16], [7], [66], [13].

Recording and Replay Overhead. In terms of recording overhead, except for fine-grained record-and-replay techniques [80], [5], [39], [70], most techniques have low overhead (e.g., less than 10%) similar to C^{2}SR. For the replay, C^{2}SR's execution reconstruction is comparable or faster than existing record-and-replay techniques, if C^{2}SR does not need to retry the reconstruction due to the timeline projection adjustment (§ IV-E). However, as shown in Table VII, C^{2}SR requires retrials to reconstruct the execution. For such cases, C^{2}SR is slower than existing techniques. For instance, C^{2}SR had to repeat 14 times to reconstruct the execution of Firefox in Table VII, leading to 14 times replay overhead than a typical system call replay technique [27], [87], [91], [21]. Note that replay acceleration [93], [52], [91], [53] and browser-specific approaches [71], [13], [16], [7], [66], [13] can replay faster than C^{2}SR while they are often not effective in reconstructing attack delivery processes.

Interactable Replay of Exploit Delivery Process. Browser-specific approaches [71], [13], [16], [7], [66], [13] can provide the interactable replay capability. However, most of them [13], [16], [7], [66], [13] aim to replay high-level web application behaviors such as mouse-events and keyboard-events. Unfortunately, they cannot reproduce exploit delivery processes that often have to replay low-level system call events. [71] replays lower-level events than others. Hence, it may record and replay some exploit delivery processes. However, it requires significant changes in browser internals.

B. Other Related Works

Network Provenance Systems. Network provenance techniques [104], [14], [103], [94] track network traffic between hosts in the same network environment to identify causal relationships across multiple hosts. C^{2}SR is complementary to such techniques such that they can be used with C^{2}SR to fully understand details of cyber attacks across multiple hosts.

Taint Analysis. Taint analysis techniques [75], [102], [18], [47], [101] track information flow between a source to a sink. Decoupled taint analysis techniques [41], [67], [68], [84] were developed to improve the run-time performance. Their idea is to decouple a target process from expensive taint analysis procedures by allocating spare cores to do the taint tracking.

Additional Forensic Techniques. As we discussed earlier, graph-based forensics analysis techniques have proposed [49], [35], [50], [54], [51], [65], [69], [10], [17], [67], [84], [28], [42], [43], however, they focus on identifying causal relations and do not allow examining details of the execution. There are efforts to reduce the space overhead of provenance data [99], [38], [55], [64]. Data reduction techniques are orthogonal to C^{2}SR such that their idea can be used to further reduce the execution trace of C^{2}SR. Recently, novel provenance inquiry techniques [59], [33], [36], [81], [98] were proposed for easier and timely investigations for advanced attacks.

VIII. Conclusion

We propose a novel technique, C^{2}SR, to enable effective cybercrime scene reconstruction by recreating an attack delivery chain from a long execution of a complex application. Its core technique is the resource based execution partitioning, that allows reproducing the attack relevant events without wasting time in reconstructing irrelevant events to the attack. Furthermore, it enables an important forensic capability, which is the interactivity of a reconstructed execution. Our evaluation results with 26 real-world applications show that it has low recording overhead (less than 5.47%), and is highly effective in reconstructing partial executions (less than 1.3% of the entire execution) of long-running applications.

ACKNOWLEDGMENTS

We thank the anonymous referees and our shepherd Adam Bates for their constructive feedback. The authors gratefully acknowledge the support of NSF (1916499, 1908021, 1850392, 1909856, and 1916500). The work was also partially supported by a Mozilla Research Award and a Facebook Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Advanced Tools for WebAssembly. https://webassembly.org/getting-started/advanced-tools/.

[3] Ashar Ahmad, Muhammad Saad, Mostafa Bassiouni, and Aziz Mohaisen. Towards blockchain-driven, secure and transparent audit logs. In *the MobiQuitous '18*.

[4] Ashar Ahmad, Muhammad Saad, and Aziz Mohaisen. Secure and transparent audit logs with blockaudit. *Journal of Network and Computer Applications*.

[5] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *SOSP '09*.

[6] Athanasios Andreou, Giridhari Venkatadri, Oana Goga, Krishna P. Gummadi, Patrick Loiseau, and Alan Mislove. Investigating ad transparency mechanisms in social media: A case study of facebooks explanations. In *NDSS'18*.

[7] Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN'11.

[8] Apache Web Server, 2019. https://httpd.apache.org/.

[9] Arora. Arora web browser, 2019. https://github.com/Arora/arora.

[10] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *the 24th USENIX Conference on Security Symposium*, SEC'15.

[11] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *NDSS'15*.

[12] Midori Browser, 2019. https://www.midori-browser.org/.

[13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *the 26th Annual ACM Symposium on User Interface Software and Technology*.

[14] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *SIGCOMM'16*.

[15] W.J. Chisum and B.E. Turvey. *Crime Reconstruction*. Elsevier, 2011.

[16] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98.

[17] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08*.

[18] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07.

[19] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security'09*.

[20] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: Investigating ad fraud in android applications. In *the 12th Annual International Conference on Mobile Systems, Applications, and Services*.

[21] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP'11*.

[22] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse debugging of failures in deployed software. In *OSDI'18*.

[23] CVE-2016-2087. https://www.cvedetails.com/cve/CVE-2016-2087/.

[24] Cycura/rtaint: Reverse taint tool. https://github.com/Cycura/rtaint.

[25] Debugging Node.js apps, 2018. https://nodejs.org/en/docs/guides/debugging-getting-started/.

[26] derjanb. A Bitcoin miner that supports pure Javscript, WebWorker and WebGL mining, 2019. https://github.com/derjanb/hamiyoca.

[27] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[28] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI).

[29] Eval Villain – Get this Extension for Firefox (en-US), 2020. https://addons.mozilla.org/en-US/firefox/addon/eval-villain/.

[30] Firefox Developer Tools, 2018. https://developer.mozilla.org/en-US/docs/Tools/.

[31] Firefox Web Browser. https://www.mozilla.org/en-US/firefox/.

[32] Association for Crime Scene Reconstruction. Crime Scene Reconstruction, 2019. https://www.acsr.org/.

[33] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *ATC'18*.

[34] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[35] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In *SOSP'05*.

[36] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam M. Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS'18*.

[37] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *ISCA'14*.

[38] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security'18*.

[39] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *PLDI'13*.

[40] InspIRCd IRC Server. http://www.inspircd.org/.

[41] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *ACM CCS'13*.

[42] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS'17*.

[43] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security'18*.

[44] Yang Ji, Sangho Lee, and Wenke Lee. Recprov: Towards provenance-aware user space record and replay. In *the 6th International Workshop on Provenance and Annotation of Data and Processes - Volume 9672*, IPAW 2016. Springer-Verlag.

[45] Shrinivas Joshi and Alessandro Orso. Scarpe: A technique and tool for selective capture and replay of program executions. *IEEE International Conference on Software Maintenance, ICSM*, 2007.

[46] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *ASIACCS'17*.

[47] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12.

[48] Wei Ming Khoo. A taint-tracking plugin for the Valgrind memory checking tool, 2020. https://github.com/wmkhoo/taintgrind.

[49] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP'03*.

[50] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *ACM CCS'10*.

[51] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F. Cretu-Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI: Modeling-based causality inference in audit logging for attack investigation. In *NDSS'18*.

[52] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07*.

[53] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In Vishal Misra, Paul Barford, and Mark S. Squillante, editors, *SIGMETRICS'10*.

[54] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS'13*.

[55] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: Garbage collecting audit log. In *ACM CCS'13*.

[56] Kyu Hyung Lee, Yunhui Zheng, Nick Sumner, and Xiangyu Zhang. Toward generating reducible replay logs. In *PLDI'11*.

[57] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS'18*.

[58] Lighttpd Web Server, 2019. https://www.lighttpd.net/.

[59] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS'18*.

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI'05*.

[61] Lynx browser, 2019. https://lynx.browser.org/.

[62] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat N. Venkatakrishnan. Propatrol: Attack investigation via extracted high-level tasks. In Vinod Ganapathy, Trent Jaeger, and R.K. Shyamasundar, editors, *Information Systems Security*. Springer.

[63] Di Ma and Gene Tsudik. A new approach to secure logging. *Trans. Storage*, 5(1), March 2009.

[64] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *ATC'18*.

[65] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security'17*.

[66] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI'10*.

[67] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *ASE'16*.

[68] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *the 24th USENIX Conference on Security Symposium*, SEC'15.

[69] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *ATC'06*.

[70] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS'06*.

[71] Christopher Neasbitt, Bo Li, Roberto Perdisci, Long Lu, Kapil Singh, and Kang Li. Webcapsule: Towards a lightweight forensic engine for web browsers. In *ACM CCS'15*.

[72] Christopher Neasbitt, Roberto Perdisci, Kang Li, and Terry Nelms. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In *ACM CCS'14*.

[73] Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahamad. Towards measuring and mitigating social engineering software download attacks. In *USENIX Security'16*.

[74] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[75] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS'05*.

[76] NGINX Web Server, 2019. https://www.nginx.com/.

[77] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *ATC'17*.

[78] Octane 2.0 JavaScript Benchmark. https://chromium.github.io/octane/.

[79] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *NDSS'20*.

[80] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *SOSP'09*.

[81] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *ACM CCS'18*.

[82] Peter Webster. Practical Crime Scene Analysis and Reconstruction. *Australian Journal of Forensic Sciences*, 42(3).

[83] ProFTPD Server. http://www.proftpd.org/.

[84] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *OSDI'16*.

[85] QupZilla browser, 2019. https://github.com/QupZilla/qupzilla.

[86] Shiru Ren, Chunqi Li, Le Tan, and Zhen Xiao. Samsara: Efficient deterministic replay with hardware virtualization extensions. In *Proceedings of the 6th Asia-Pacific workshop on systems*.

[87] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *ATC'16*.

[88] Sherif Saad, Farhan Mahmood, William Briguglio, and Haytham Elmiligi. Jsless: A tale of a fileless javascript memory-resident malware. *ArXiv*, abs/1911.11276, 2019.

[89] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. Continuous tamper-proof logging using tpm 2.0. In *International Conference on Trust and Trustworthy Computing*.

[90] Speedometer 2.0. https://browserbench.org/Speedometer2.0/.

[91] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *ATC'04*.

[92] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Decoupling provenance capture and analysis from execution. In *the 7th USENIX Conference on Theory and Practice of Provenance*, TaPP'15.

[93] Dinesh Subhraveti and Jason Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS '11*.

[94] Benjamin E. Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H. Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *ACM CCS'18*.

[95] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. Enabling reconstruction of attacks on users via efficient browsing snapshots. In *NDSS'17*.

[96] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *ASPLOS'11*.

[97] Websites use your CPU to mine cryptocurrency even if you close them. https://www.hackread.com/websites-use-cpu-mine-cryptocurrency-even-close/.

[98] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security'17*.

[99] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS'16*.

[100] Attila A Yavuz, Peng Ning, and Michael K Reiter. Efficient, compromise resilient and append-only cryptographic constructions for digital forensics. *computer*, 15(29):37–43, 2011.

[101] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM CCS'07*.

[102] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI'06*.

[103] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *SOSP'11*.

[104] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *ACM SIGMOD'10*.

[105] zlib Home Site, 2017. https://zlib.net/.

APPENDIX

A. The Complete List of System Call Categorization

As shown in Table VIII , we categorize Linux syscalls that access external resources into three resource types; 1) Random-Access, 2) Sequential-Access, and 3) Timing Dependent.

Random-Access Resources (125 syscalls). The *random-access* type syscalls mostly access file systems, networks, and internal system states. They are considered as external resources because they may be different between machines and those resources may not exist during the reconstruction.

Sequential-Access Resources (23 syscalls). The *sequential-access* type syscalls access networks and pipes that are considered as an external resource and cannot be accessed randomly.

Timing-Dependent Resources (11 syscalls). The *timing dependent* type syscalls are accessing date, time, or clock. Reading from random devices (e.g., `/dev/random`) is also categorized into this type.

Syscalls that are not recorded (156 syscalls). There exist syscalls that we do not record because their executions do not contribute cybercrime scene reconstruction. C^{2}SR aims to reconstruct a deterministic (exploit delivery) execution. The execution of syscalls in this category do not affect deterministic (exploit delivery) executions. For instance, the order of signals (e.g., accomplished by rt_sig* syscalls) does not affect the successful exploit delivery (if it is deterministic). Our focus is not a faithful replay of non-deterministic events that are irrelevant to the successful execution (i.e., exploitation process). If reconstruction of non-deterministic exploit execution (e.g., concurrency attacks) is out of scope. To do so, one must leverage faithful record-and-replay techniques. Note that most exploits in practice are fairly deterministic (they mostly succeed), as otherwise, the attacks would mostly fail. We do not encounter such cases in our experiments, however, we assume that the current version of C^{2}SR may not be able to handle corner cases if a non-deterministic event triggered by not-recorded syscalls.

TABLE VIII. LIST OF LINUX SYSCALLS WITH THE RESOURCE TYPE CATEGORIES.

Type	Syscall List
Random Access	read, write, ioctl, pread64, pwrite64, readv, writev, sys_preadv, sys_pwritev, open, close, stat, fstat, lstat, lseek, access, setitimer, sendfile, getsockname, setsockopt, getsockopt, uname, fcntl, flock, fsync, fdatasync, truncate, ftruncate, getdents, getcwd, chdir, fchdir, rename, mkdir, rmdir, creat, link, unlink, symlink, readlink, chmod, fchmod, chown, fchown, lchown, umask, getrlimit, getrusage, sysinfo, getuid, getgid, setuid, setgid, geteuid, getegid, setpgid, getppid, getpgrp, setsid, setreuid, setregid, getgroups, setgroups, setresuid, getresuid, setresgid, getresgid, getpgid, setfsuid, setfsgid, getsid, capget, capset, mknod, ustat, statfs, fstatfs, sysfs, pivot_root, _sysctl, setrlimit, chroot, sync, acct, mount, umount2, sethostname, setdomainname, quotactl, setxattr, lsetxattr, fsetxattr, getxattr, lgetxattr, fgetxattr, listxattr, llistxattr, flistxattr, removexattr, lremovexattr, fremovexattr, getdents64, utimes, openat, mkdirat, mknodat, fchownat, futimesat, newfstatat, unlinkat, renameat, linkat, symlinkat, readlinkat, fchmodat, faccessat, splice, sync_file_range, vmsplice, utimensat, timerfd_settime, timerfd_gettime, prlimit64, name_to_handle_at, open_by_handle_at
Sequential Access	read, write, ioctl, pread64, pwrite64, readv, writev, sys_preadv, sys_pwritev, pipe, select, connect, accept, sendto, recvfrom, sendmsg, recvmsg, listen, getpeername, accept4, pipe2, recvmmsg, sendmmsg
Timing Dependent	settimeofday, gettimeofday, times, utime, time, clock_settime, clock_gettime, read (on random devices), pread64 (on random devices), readv (on random devices), sys_preadv (on random devices)
Not Recorded	poll, mmap, mprotect, munmap, brk, rt_sigaction, rt_sigprocmask, rt_sigreturn, sched_yield, mremap, msync, mincore, madvise, shmget, shmat, shmctl, dup, dup2, pause, shutdown, clone, fork, vfork, execve, exit, wait4, kill, semget, semop, semctl, shmdt, msgget, msgsnd, msgrcv, msgctl, ptrace, syslog, rt_sigpending, rt_sigtimedwait, rt_sigqueueinfo, rt_sigsuspend, sigaltstack, uselib, personality, getpriority, setpriority, sched_setparam, sched_getparam, sched_setscheduler, sched_getscheduler, sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval, mlock, munlock, lockall, munlockall, vhangup, adjtimex, swapon, swapoff, reboot, init_module, delete_module, readahead, tkill, futex, sched_setaffinity, sched_getaffinity, set_thread_area, get_thread_area, lookup_dcookie, epoll_create, remap_file_pages, set_tid_address, restart_syscall, semtimedop, fadvise64, timer_create, timer_settime, timer_gettime, timer_getoverrun, exit_group, epoll_wait, epoll_ctl, tgkill, mbind, set_mempolicy, get_mempolicy, mq_open, mq_unlink, mq_timedsend, mq_timedreceive, mq_notify, mq_getsetattr, kexec_load, waitid, add_key, request_key, keyctl, ioprio_set, ioprio_get, inotify_init, inotify_add_watch, inotify_rm_watch, migrate_pages, pselect6, ppoll, unshare, set_robust_list, get_robust_list, move_pages, epoll_pwait, signalfd, timerfd_create, eventfd, fallocate, signalfd4, eventfd2, epoll_create1, dup3, inotify_init1, rt_tgsigqueueinfo, fanotify_init, fanotify_mark, clock_adjtime, syncfs, setns, getcpu, kcmp, finit_module, getpid, socketpair, gettid, process_vm_readv, process_vm_writev, modify_ldt, iopl, ioperm, socket, bind, prctl, arch_prctl, io_setup, io_destroy, io_getevents, io_submit, io_cancel, tee, perf_event_open, nanosleep, getitimer, alarm, timer_delete, clock_getres, clock_nanosleep

Library Calls. Recall that C^{2}SR also records user inputs such as keystrokes. To do so, we interpose X-Window libraries and implement record-and-replay mechanism via library hooking. C^{2}SR treats such library calls as same as sequential access resources except for that C^{2}SR does not raise exceptions if there is any new inputs in a reconstructed execution.