Only Pay for What You Leak: Leveraging Sandboxes for a Minimally Invasive Browser Fingerprinting Defense

Ryan Torok
Department of Computer Science
Princeton University
Princeton, New Jersey, USA
rtorok@princeton.edu

Amit Levy
Department of Computer Science
Princeton University
Princeton, New Jersey, USA
aalevy@princeton.edu

***Abstract*—We present Sandcastle, an entropy-based browser fingerprinting defense that aims to minimize its interference with legitimate web applications. Sandcastle allows developers to partition code that operates on identifiable information into sandboxes to prove to the browser the information cannot be sent in any network request. Meanwhile, sandboxes may make full use of identifiable information on the client side, including writing to dedicated regions of the Document Object Model. For applications where this policy is too strict, Sandcastle provides an expressive *cashier* that allows precise control over the granularity of data that is leaked to the network. These features allow Sandcastle to eliminate most or all of the noise added to the outputs of identifiable APIs by Chrome's Privacy Budget framework, the current state of the art in entropy-based fingerprinting defenses. Enabling unlimited client-side use of identifiable information allows for a much more comprehensive set of web applications to run under a fingerprinting defense, such as 3D games and video streaming, and provides a mechanism to expand the space of APIs that can be introduced to the web ecosystem without sacrificing privacy.**

***Index Terms*—browser fingerprinting, information entropy, sandboxing, k-anonymity**

1. Introduction

The web has become an increasingly capable and diverse platform. New HTML, CSS, and JavaScript standards have allowed web browsers to run applications that, in many cases, are as powerful and feature-full as native applications. The web ecosystem today enables applications to perform increasingly diverse tasks including 3D graphics, real-time video, and access to USB devices.

Unfortunately, these features also enable third-party actors, such as advertising networks, to circumvent browser protections against global tracking [1], [2] with fine-grained fingerprinting. Fingerprinting allows a web page to identify users using a collection of browser and device characteristics exposed via browser APIs, and share that fingerprint with an interested server. For example, the way in which pixels are drawn to in a HTML5 canvas element can identify the a specific GPU make and model, WebRTC APIs expose a device's internal IP address, etc. Though any single fingerprinting vector cannot, typically, uniquely identify a user across web sites, attackers can generate precise fingerprints by combining them.

Defending against fingerprinting is challenging. Legitimate applications use identifying web APIs, so removing such APIs outright would preclude many useful applications. Recent proposals, such as Chrome's Privacy Budget [3], [4], allow access to APIs but suggest limiting the total entropy an application can learn through these APIs. Unfortunately, such proposals face two major challenges. First, the browser vendor must determine the entropy associated with each API. However, evolving properties such as timing and precision make this a dynamic target that might change between versions of the same browser. Second, this approach still precludes many legitimate applications that make extensive use of identifying APIs but share little or no entropy learned from those APIs.

In essence, both categories of defenses punish applications for their access to identifying APIs, rather than for the entropy they actually leak.

In this paper, we present Sandcastle, a novel fingerprinting defense that aims to permit legitimate web application behaviors that must be excluded by traditional defenses. Sandcastle relies on two key observations. First, if an application reads identifying data but never leaks it to the network, that data could not have contributed to fingerprinting. Second, when data does leak to the network, only the amount of entropy actually leaked contributes to fingerprinting. For example, an application might read the list of connected USB peripherals (which contains a lot of entropy), but only leak whether it has seen *any* peripherals—contributing very little entropy to fingerprinting.

Contrary to prior approaches, Sandcastle adopts a *pay for what you leak* policy. Applications can access data from identifying APIs as long as Sandcastle can ensure that data does not leak to the network. When applications *do* leak data, the data is charged to an entropy budget based on the amount of entropy leaked, not the amount of entropy accessed. As a result, trackers are thwarted because they will not be permitted to leak sufficient entropy to identify

a user, while still permitting many legitimate applications. Applications that do not leak fingerprintable information outside the browser do not suffer, and applications that do are only charged for the amount they actually leak.

Sandcastle provides an API for sandboxing code that enables applications to perform unlimited computation using sensitive data on the client side, including access a controlled DOM. Moreover, Sandcastle provides an expressive interface to finely control the granularity of data leaked outside the sandbox. These features allow applications to tightly bound the quantity of sensitive information they expose, enabling entire classes of applications that would be blocked by previous defenses, like HD games and video streaming, to run under a fingerprinting defense.

To achieve these goals, Sandcastle addresses a number of technical challenges. Due to the dynamic nature of the DOM and its interactions with CSS styling, data can leak to the network through a variety of covert channels. For example, changing the size of a DOM element can implicitly affect which CSS style rules are evaluated and thus which background image is fetched for another element. Moreover, Sandcastle allows arbitrary JavaScript to compute over identifying data, potentially exposing a high-bandwidth timing channel. Finally, Sandcastle must determine how much entropy is actually leaked by the application without a priori knowledge of the application behavior.

Sandcastle addresses these three challenges with careful design of the sandbox. In order to control the entropy leaked from a sandbox, Sandcastle requires applications to specify the space of results sandboxes may produce and time quanta at which those results may become available. To prevent covert channels through the DOM, Sandcastle leverages the browser's same-origin policy to provide DOM regions multiple sandboxes can share.

This paper provides background and motivating examples on fingerprinting attacks and defenses, describes the design of Sandcastle and its prototype implementation in the Chromium browser, and shows experimentally that Sandcastle's guarantees allow the browser to achieve significant noise reduction over the state-of-the-art with minimal performance overhead.

2. Background and Related Work

Browser fingerprinting attacks allow a server to uniquely identify the device or browser installation accessing the service without using any methods the browser deliberately exposes for that purpose, such as session cookies. Academics [5] and journalists [6] have found that several advertising networks use browser fingerprinting to track users across web pages without using third-party cookies.

Fingerprinting attacks generally involve combining many sources of browser information together to produce a fingerprint with fine enough granularity to identify individual devices. For example, while the browser vendor, GPU model, available memory, etc, each identify devices at only a very coarse granularity, combining such data can identify an individual device with high specificity.

Moreover, in order to be useful for tracking, fingerprinting scripts aim to collect data that not only has high entropy, but also stable over time, allowing the server to reliably re-identify the same device. Therefore, the best fingerprinting surfaces reveal properties of the machine the browser is running on. Attacks that identify the type of CPU or GPU on the system, the capacity of the system memory, or the operating system the browser is running on are powerful assets for the attacker, since this information is unlikely to change over time and even persists across different browsers on the same device [7].

Previous work has identified several sources of fingerprintable information in web browsers. After Mowery and Shacham demonstrated the ability to fingerprint the user's GPU using the HTML canvas in 2012 [7], the community has since added local IP addresses, WebRTC device IDs, WebGL graphics, enumerating fonts [8], WebAudio channel data, CPU and memory specs [9], and even peripheral timestamps [10], among others. As these attack surfaces continue to be discovered, trackers continually aggregate them into JavaScript libraries like fingerprint.js [9], significantly lowering the barrier to large-scale tracking.

Fingerprinting surfaces generally fall into one of two categories, called *passive* and *active* fingerprints [3], [5]. Passive fingerprints are information generated by the browser for its communication tasks, such as HTTP request metadata, security headers, and network addresses. Meanwhile, active fingerprints are leaked by browser APIs that untrusted website code can obtain and leak through network requests.

In this paper, we focus on active fingerprints. While the entropy budget concept Sandcastle uses applies to passive fingerprints as well, the handling of passively fingerprintable data is always performed by trusted browser code, meaning the browser can trivially understand how much sensitive data the server receives. Active fingerprinting is more interesting for our purposes, because there is a fundamental disparity between what untrusted website code *might do* and what it *actually does*. For a robust defense, the browser must assume sensitive data leaked by its APIs is used for tracking unless it can prove otherwise. Our solution allows applications to bridge this gap by guaranteeing to the browser they access this data safely.

In response to the use of these attacks in the wild, defenses against fingerprinting have recently been of particular interest in both the academic and browser development communities. The most comprehensive approach is to aggressively block access to sensitive information. Privacy-centric browsers including Firefox [11], Brave [12], and particularly Tor Browser [13] have all employed this strategy to varying degrees. However, this strategy comes with the serious drawback that it blocks entire classes of applications that make legitimate use of the sensitive APIs and degrades the user experience in general.

For example, consider the screenshot of Tor Browser in Figure 1. To prevent an attacker leaking the screen size, Tor Browser rounds the window width and height down to a multiple of 100 pixels and fills the remaining space with gray borders. While this mitigates use of the window size

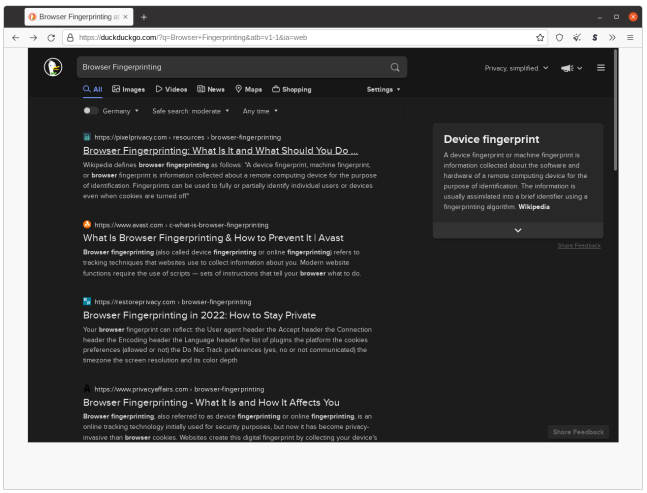


Figure 1: Tor Browser rounds the display size down to 100-pixel increments to prevent websites from reading the screen size, filling the remaining space with gray borders [14].

for fingerprinting, it also degrades the user experience for most websites, which do not use it for fingerprinting.

More recently, fingerprinting defenses have aimed to address these usability issues. Iqbal et al. [15] proposed using machine learning to detect fingerprinting patterns, though this is fundamentally a best-effort system and does not intend to provide the same level of robustness as much of the techniques we describe in the remainder of this section.

For a strong combination of flexibility and robustness, existing browsers have generally adopted two major defense patterns. Brave recently deployed *fingerprint randomization*, which reduces the entropy of sensitive APIs by introducing random noise [16]. Brave's development team also highlighted a proposal to use a depleting entropy budget to limit the amount of sensitive information the website can leak while allowing limited use of sensitive APIs. [17]. Chrome's Privacy Budget proposal [3] suggests using both.

Unfortunately, both of these strategies run into practical issues in their current forms. Introducing noise can still be harmful to the application and is really only practical for a few libraries like WebGL and WebAudio, where the browser can be relatively sure the noisy data will not harm the application, and solutions like PriVaricator [18] and FPRandom [19] that use smart randomization to break linkability between site visits cannot circumvent this issue for APIs that have deterministic specifications. Even with smart randomization, an online scheduling app would be unwilling to tolerate even a small chance of displaying the wrong time zone, for example.

As for the entropy budget strategy, both Brave's and Chrome's development teams acknowledge entire classes of applications like 3D games and audio analysis apps, that, with any reasonable amount of noise, simply leak too much information to be accommodated by the defense. Frustratingly, the servers of each of these legitimate applications require little to none of the sensitive data the applications access, but current browsers are not smart enough to understand that this usage pattern cannot possibly cause a fingerprinting attack.

This is precisely the issue Sandcastle addresses with its *pay for what you leak* policy. Because Sandcastle allows developers to sandbox components of code that operate on sensitive data on the client-side and only charges data leaked, not data accessed, it enables entire classes of applications to run under an entropy budget defense that would otherwise be blocked or be served data so noisy they could not operate.

One other type of fingerprinting studied heavily in previous work is *extension fingerprinting* [9], [20]–[23], wherein the attacker enumerates the browser extensions the user has installed by serving DOM elements that tend to be modified by extensions, such as banner advertisements that are deleted by ad blockers. We consider these attacks out of scope for Sandcastle, since they do not involve sensitive APIs, but previous solutions like Simulacrum [24], which implements a parallel DOM for extensions, and CloakX [25], which diversifies client-side identifiers, have demonstrated working defenses for these attacks that could be deployed alongside Sandcastle. We also observe that extension fingerprinting may not be incompatible with sandboxing, which future work could leverage to allow Simulacrum's strategy to run with lower overhead.

3. Motivating Examples

The design of Sandcastle is based on two high-level goals for the future of the web. First, we desire to allow as many legitimate applications that run without fingerprinting defenses to remain deployable with as little interference as possible. Second, we aim to enable richer applications by broadening the space of APIs web browsers can safely expose. In this section, we highlight some examples of applications that motivate these goals.

3.1. Resource-Intensive Online Game

The broadening feature set of the web and rising bandwidth of the internet has led to the web becoming a popular platform for gaming applications. However, modern video gaming platforms tend to be rather resource-intensive, requiring high CPU, memory, and GPU capacity in order to perform their physics calculations and HD graphics rendering at high frame rate. Accordingly, gaming platforms often desire to check if a player's machine has the necessary hardware resources to run the application before performing the bandwidth-intensive task of downloading the game assets.

These types of hardware checks are by no means unique to gaming applications. When ETS moved the GRE and TOEFL exams online during the COVID-19 pandemic, the web platform hosting the exam performed hardware compatibility checks before administration [26]. While testing platforms are generally much less resource-intensive than a modern game engine, the hardware checks were nonetheless employed by the test administrators, since an incompatibility

in this setting can have high cost; it can lead to the forfeiture of the student's test progress and fees.

Unfortunately, a traditional entropy budget defense significantly limits the scope of these checks, because the user's hardware specs are a high-entropy source of fingerprintable information. Even though the server of such an application needs only a single bit of information conveying whether the user's hardware is compatible, the defense blocks the checks because it cannot distinguish between information that can be leaked to the network and information that is only used on the client side. Meanwhile, Sandcastle allows this type of application to perform unlimited sensitive hardware checks inside its non-network-accessible sandbox without being charged for the entropy or suffering from noise. Using this structure, the sandbox can display the exact incompatibilities found, if any, to the user using Sandcastle's DOM sandboxes, but only leak and be charged for the single bit of entropy the server requires.

3.2. High-Performance Video Streaming

Video streaming currently makes up about 75% of current internet traffic [27], and designing advanced algorithms to determine the highest possible resolution, or bit rate, supported by the user's network capacity has seen a great deal of academic work in the past several years [27]–[31]. These Adaptive Bit Rate (ABR) algorithms must balance several factors such as the buffer size, the network bandwidth, and, for ABR schemes like NAS [30] [31] that use superresolution to locally improve the video quality, the available compute capacity.

Furthermore, experimental data from this space indicates that ABR algorithms are quite fragile [29], where a single poor bit rate decision often causes a cascading effect of poor resource usage that can significantly degrade the quality of experience (QoE) [32] for the user. Likewise, operating on noisy data about the available hardware resources is unlikely to obtain satisfactory streaming performance. Similar to the previous example, Sandcastle accommodates this use case with significantly reduced noise over existing randomization techniques, because the browser is assured that only the bit rate, chosen from a small set of options, is leaked to the network, not the high-entropy data the algorithm uses to decide on a bit rate.

3.3. System Resource Monitor

Finding all the hardware specifications of a machine and examining its performance data often requires navigating several native applications, and the procedure varies wildly across different operating systems. Deploying such a system on the web would be much more streamlined, but this is an example of an application that depends on APIs that, for good reason, do not exist at all in today's browsers. APIs that leak this type of detailed information about the system, such as the CPU and memory usage across different processes, would expose a plethora of identifiable information and would be recklessly unsafe to deploy without an entropy-based fingerprinting defense. Meanwhile, Sandcastle makes this type of application possible and safe because all of the information displayed in the DOM remains inside of the sandbox and cannot be used for fingerprinting. More generally, Sandcastle enables the browser to provide an execution environment in which applications can perform useful work with APIs that are entirely unsafe to deploy in today's browsers.

4. Sandcastle

Sandcastle is a novel fingerprinting defense that allows web applications unlimited access to sensitive APIs inside a sandbox. The sandbox is allowed to perform arbitrary computation on data returned by APIs flagged as sensitive without depleting its entropy budget and access sandboxed DOM elements. However, it is unable to make network requests or access program variables owned by the calling context. Instead, it communicates with the calling context through a narrow, prespecified interface. The calling context is charged for the entropy leaked through that interface. In this section, we discuss the challenges of building a practical and robust fingerprinting defense and describe Sandcastle's design and implementation.

4.1. The Sandbox API

Unlike existing entropy budget strategies [3], [4], [17], Sandcastle does not charge the entropy of a secret to the entropy budget when it is returned from a browser API. Instead, it asks the developer to specify possible *exit states* of the sandbox. Upon returning from the sandbox, Sandcastle reduces the remaining entropy budget minimum of the entropy of the *exit states* and the entropy of APIs used inside the sandbox. Sandboxes also enforce strong isolation from each other and their calling context.

Example 1 shows a diagram of the sandbox API. To invoke a sandbox, the programmer must specify the possible return values the sandbox may produce and the time quanta at which it may return. Together, these are the sandbox's exit states. The caller may also include an opaque origin ID, which enables JavaScript context sharing as well as partitioning access to sandboxed DOM elements (see Section 4.2). Finally, the developer provides a function, and arguments to the function, to evaluate inside the sandbox. After the sandboxed function completes, Sandcastle restricts return values to those specified in the exit states or `undefined` (to cover all other return values).

Arguments to and return values from the sandboxed function can be of any type that can by copied across the sandbox isolation boundary. This is implementation dependent (Section 5) but generally includes most JavaScript objects, but not functions [33].

Using the coarse isolation granularity of a sandbox allows Sandcastle to control entropy leaked through *timing channels*. Because Sandcastle permits the application to run arbitrary code inside the sandbox, an attacker can express

Example 1: The Sandbox API

```
var r = sandcastle.sandbox(
    [true, false],// Possible return values
    [10, 20, 30], // Possible return times (ms)
    "opaque_1",   // Opaque origin ID
    function(arg1, arg2) { // Sandbox code
        return arg1 + arg2 < someSecret();
    }
    1, 2          // Arguments
);
```

additional entropy beyond the return value entropy by varying the sandbox's exit time. For example, the sandbox could sleep for a secret-dependent length of time, allowing unsandboxed code to infer the secret by measuring the sandbox's execution time. Worse, browsers cannot easily bound the entropy of these timing channels. While browser timing APIs have historically introduced some noise to prevent other side-channel attacks, the attacker can cope with this noise by decreasing the granularity of their sleep.

Sandcastle handles this problem by requiring sandboxes to list the time quanta at which they may return. When a sandboxed function completes, Sandcastle waits until one of the remaining exit times before returning. If the longest provided interval is reached before the function completes, the sandbox aborts and a timeout is returned[1] By requiring sandboxes to be explicit about their ability to cause timing channels, we transform the high-bandwidth space of return times to a small discrete set that carries developer-chosen entropy. Often, the developer can choose a conservative but reasonable quantum leaking only a single bit—that the function indeed returned.

One consideration for Sandcastle's API consumers is how to choose an exit time, when the website code may run on heterogeneous hardware with different processing speeds. If no times are specified in the API, Sandcastle uses a default execution time of 20 ms, which is sufficient for non-time-sensitive applications. However, high-performance apps that utilize many sandboxes may be unsatisfied with this overhead. In cases where performance matters, the developer can choose to perform a lightweight benchmark of the hardware at the start of their application that leaks a small amount of entropy.

For instance, if the developer possesses a benchmarking function designed to run for 1 ms on a 5 GHz processor, she could sandbox the function with the exit time list [1.00, 1.11, 1.25, 1.43, 1.67, 2.00, 2.50] (all times in ms) to profile the CPU from 5 GHz down to 2 GHz in 0.5 GHz increments. Though this strategy charges a few bits to the application's entropy budget up front (3 bits in this example, since a timeout is a hidden eighth exit time), it eliminates the need to list multiple exit times on each sandbox the application spawns, saving the application entropy in the long run.

Another related concern is estimating performance time in the face of JIT compilation. Previous work has shown that existing JavaScript engines provide poor timing guarantees due to their complex methods for determining when and how much to optimize code [34]. Sandcastle does not provide any specific APIs dedicated to profiling JIT optimization. However, because sandboxes allow their exit times to be chosen *at runtime*, they still provide developers useful options if they know they will run a high-performance function inside a sandbox many times. First, they could convince the browser to optimize the function up front, say, by repeatedly running it at the start inside a sandbox that leaks no entropy. Alternatively, they could occasionally probe their sandboxes for faster times at fixed intervals, and shift to the faster time as the default if the sandbox returns in time. This second method does not require any profiling up front, but makes it more challenging for the developer to bound the amount of entropy spent profiling. Which side of this tradeoff to prefer is specific to the needs of individual applications, depending on how much entropy they expect to spend for non-profiling work and the importance of fast startup times.

1. This behavior is only guaranteed in the noise model's default error handling mode. For more details, see Section 4.4.

4.2. Sandboxing DOM Elements

One of Sandcastle's major goals is to allow the web developer to avoid the cost of leaking sensitive data when it is only used on the client side. Likewise, it is strongly desirable to allow sandboxed code to write sensitive data to Document Object Model (DOM) elements without incurring any cost. This capability enables Sandcastle to guarantee privacy for applications that use advanced web features like HTML canvases and WebGL graphics, including 3D games and HD video rendering — use cases the original Privacy Budget proposal references as *impossible* to support [4].

Meanwhile, the requirement to allow sandboxing DOM elements poses an interesting problem: how can we allow the application to safely write sensitive data to the DOM without allowing unsandboxed code to read and leak it? Naively, we could attempt to annotate individual DOM objects with the secrets obtained by the sandboxes that write to them. However, this strategy is insufficient, because modifications to DOM objects that change their size can affect the position and size of neighboring objects. If Sandcastle only considered secrets attached to the object that is actually read, an attacker could encode a secret value in the size of a DOM object, then leak the secret by reading, say, the `offsetLeft` attribute of the sibling object.

Further, recent advances in CSS styling such as flexible boxes [35] and lazy loading of resources based on the viewport [36] make maintaining a mapping between DOM elements and secrets an extremely difficult task. While we cannot categorically determine such a strategy is impossible, our analysis of this direction indicates that as flexible and innovative layout and styling options continue to be developed for the web, the implementation of such a defense would be prohibitively impractical to maintain, if not just incorrect.

Instead, Sandcastle requires that all DOM objects created or modified by sandboxes must live inside a dedicated `<iframe>` element the browser has special awareness of, which we call a DOM sandbox. While this strategy does

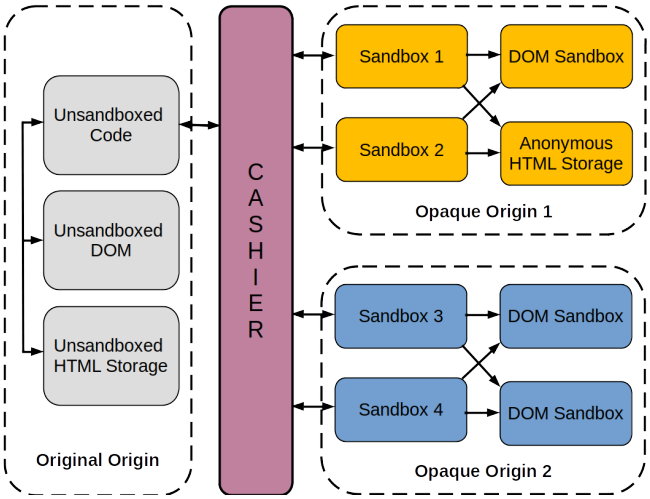


Figure 2: Partitioning the DOM using opaque origins and DOM sandboxes, which are based on anonymous iframes.

prevent a few specific use cases like integrating flex layouts across the sandbox boundary, DOM sandboxes greatly simplify our DOM management scheme, and also allow us to leverage a relatively new addition to the browser privacy space: anonymous iframes [37], [38].

Anonymous iframes are a special type of HTML iframe that was recently standardized to support flexible resource loading under the cross-origin embedder policy (COEP) [39], which is beyond the scope of this paper. What is important for our purposes is that, under normal circumstances, the browser's *same-origin policy* allows an iframe's DOM to be accessed by its parent only if the iframe and the parent were served by the same website, or origin. Meanwhile, an anonymous iframe prohibits access from its parent, even if such an access would normally pass under the same-origin policy [37], [38]. Thus, Sandcastle can allow sandboxes to write sensitive data to the DOM in anonymous iframes and guarantee the data is hidden from unsandboxed code. Privacy is protected by the same-origin policy!

To maximize flexibility of the sandboxed DOM, including access to HTML5 storage, both code sandboxes and DOM sandboxes can be assigned to an *opaque origin*[2] [41]. As Figure 2 illustrates, opaque origins allow the developer to specify partitions of code and DOM elements that share a collection of sensitive data. Opaque origins allow sandboxed regions of the application to guarantee to Sandcastle that any sensitive data observed in one opaque origin cannot be viewed by another, which, as we will see in the next section, can allow the cashier to charge less entropy when a leak occurs. Sandcastle also automatically shares the same JavaScript context between sandboxes that use the same opaque origin, which simplifies implementation of global state and event handling inside the sandbox without affecting the cashier's ability to reason about the secrets accessed.

2. Opaque origins on anonymous iframes are currently in limbo in the W3C proposal due to anticipated difficulties in standardizing the behavior of opaque access to storage APIs [37], [38]. If this constraint remains, future deployments of Sandcastle may be limited to a single anonymous opaque origin per page context, and the remainder of the opaque origins could use sandboxed iframes [40], which already support opaque origins, but not storage APIs.

4.3. The Cashier: Charging for Secrets

Many applications use identifying APIs but never leak their results to the network. However, some legitimate applications, for example video streaming, do send small amounts of sensitive data over the network. Sandcastle's API permits this by charging for entropy based on the minimum of the exit states' entropy and the entropy of APIs used inside the sandbox. Accounting for this leaked information is delegated to a feature we refer to as the *cashier*. The cashier's responsibility is to ensure that any entropy leaked to unsandboxed code is charged to the context's entropy budget while aiming to charge as little entropy to the budget as possible.

If the programmer had perfect knowledge of how much entropy each secret contains at all times, optimally upholding this invariant is a simple task, because the cashier would need only consider the entropy of sandboxes' spaces of exit states. Then, the programmer could decide whether or not to sandbox an access to a secret by comparing the secret's entropy with the granularity she needs. However, requiring the programmer to understand the entropy of secrets is error-prone and complicates development. Worse, it is not straightforward to abstract this decision into a third-party JavaScript library, because a browser API that reveals the entropy remaining in a secret can *itself* leak entropy, since a previous sandbox could have chosen to access a secret based on branches depending on other secrets.

Likewise, Sandcastle's cashier does not assume sandboxes access more entropy than their spaces of exit states convey, and aims to prove the lowest safe cost in both cases. One important factor in this goal is to avoid charging any piece of sensitive data more than once when it can prove it is the same data, which is a complex task. In this section, we describe at a high level how the cashier is designed and the problems it addresses, and we provide a detailed formal definition in Appendix A.

Example 2: This sandbox operates on two secrets, but the cashier cannot determine how many bits of each secret the return value carries.

```
sandcastle.sandbox(7, [10, 20], function() {
    var A = secret_a();
    var B = secret_b();
    var r = ...; // Do something with A and B.
    return r;
});
```

The Cashier Uncertainty Principle Consider the code in Example 2. This sandbox calls into two functions `secret_a`, and `secret_b`, producing two secret values, which we will call A and B. Suppose A and B each begin with 8 bits of entropy. If we ignore error states for now,

this sandbox produces an 8-bit exit state, seven from the return value and one from the time quanta. However, what is it 8 bits *of* — 8 bits of A? 8 bits of B? 4 bits of each? Unfortunately, without any knowledge of what the sandboxed function does, it is not clear from the outside which secret to deplete the 8 bits from.

Even worse, this information is critical for the cashier to know exactly; otherwise, an attacker can leak bits without the cashier's knowledge. For example, suppose after running the sandbox in Example 2 the cashier lazily depleted 4 bits from each secret. This seems safe, because the entropy charged equals the entropy leaked. However, this is not safe enough. Suppose the attacker had set `r = A` in the sandbox, making the true value leaked represent all 8 bits of A and 0 bits of B. Then, the attacker could spawn another sandbox that leaks all 8 bits of B. Now, because the cashier believes 4 bits of B were already charged by the first sandbox, it thinks charging only the remaining 4 bits is sufficient. If it does, the attacker will learn all 16 bits of A and B, but only be charged 12 bits total!

As it turns out, this strategy will *never* work unless the cashier knows exactly how much of each secret the exit state is comprised of. However, because untrusted JavaScript cannot be safely reasoned about while upholding our side channel defenses, the amount of entropy to deplete from each secret is invisible to Sandcastle. We refer to this phenomenon as the *cashier uncertainty principle*.

Embracing the Uncertainty: Saturating Secret Sets At first glance, it seems the cashier uncertainty principle would make it impossible for the cashier to make any nontrivial cost improvements. However, if we make the cashier expressive enough to record and reason about its uncertainty about which secrets are responsible for leaked bits, we can still make powerful cost optimizations!

To do this, the cashier utilizes a technique of *set saturation*. In Example 2, since the cashier does not know how many bits of each secret comprise the return value, it instead attributes 8 bits to the set, $\{A, B\}$. Then, suppose the cashier later encounters another sandbox that leaks 12 bits after accessing A and B. In this case, the cashier can say something nontrivial about the cost, because no matter what, the set $\{A, B\}$ only had 16 bits of entropy at the start. Thus, even without knowing how much of each secret contributed to the 8 bits leaked by the first sandbox, it knows it does not need to charge A and B more than 16 bits in total, so it is safe to charge only 8 bits for the second sandbox instead of the full 12. At this point $\{A, B\}$ has been charged its full entropy, and future sandboxes accessing that pair of secrets can return for free. Here, we say $\{A, B\}$ is *saturated*.

The cashier can also make strong guarantees across different sets of secrets. For example, saturating $\{A, B\}$ as above implies A and B are individually saturated as well, so sandboxes operating on A or B alone also incur no cost. Further, it can be useful to consider sets that have not been explicitly charged. Suppose A, B, and C are 8-bit secrets, and the cashier has already charged 8 bits to each of $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$. This implies that $\{A, B, C\}$ is saturated, since all 24 of its bits were spent on its subsets, even though no sandbox actually accessed all three secrets together. This would allow, say, A alone to be leaked for free, even though neither of $\{A, B\}$ or $\{A, C\}$ is saturated.

Finding the Optimal Set Using this saturation principle, the goal of Sandcastle's cashier algorithm is, for a set S of secrets being leaked, to find a superset of S that is closest to saturation (having the fewest bits left to spend), and charge the minimum of this saturation distance and the entropy leaked through the sandbox's exit state. While searching all supersets can potentially be an enormous search space, we optimize the algorithm to consider only sets that have a chance to guarantee the lowest safe cost of S. At a high level, this is achieved by considering a graph of all secrets the cashier has seen, with edges connecting secrets that could have been accessed together. Using the secret graph, sets that contain secrets disconnected from any secret in S can be disregarded, as can sets that *break chains* between one or more of their elements and S. That is, a cashier that has seen, say, the sets $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$, when calculating the cost of $\{A\}$, should consider supersets such as $\{A, B\}$ and $\{A, B, C, D\}$, but may ignore sets like $\{A, B, D\}$ because D's only connection to A was through C, which was excluded from the set. We prove these optimizations are sound (and optimal) in Appendix A.

4.4. Noisy Exit States

While Sandcastle's main goal is to reduce the amount of noise that must be added to the data leaked outside of its sandboxes to maintain privacy, there exist some usage patterns where it cannot be entirely eliminated. In existing entropy budget defenses, optimally introducing noise is challenging because the perturbation to apply to the return value in order to introduce the desired noise depends on the space of return values the API actually returns [42]. For example, if a sensitive API returns either of the string values `"foo"` or `"bar"`, then a naive character perturbation that transformed `"foo"` into `"fon"` or `"fop"` would prove almost entirely useless at adding noise.

Meanwhile, because the possible exit states from Sandcastle's sandboxes are defined by the programmer, Sandcastle can adopt a universal noise model that works for all sandboxes. The model's goal is to steadily increase the noise added to exit states as the entropy budget depletes, and, for each sandbox, produce a probability distribution over all of its possible exit states with Shannon entropy [43] equal to the desired noise. That noise entropy may then be subtracted from the entropy the exit state would leak without noise.

Aggregating Noise over Time For starting budget B, Sandcastle determines the quantity of real entropy to be leaked for the ith *logical bit* using the exponential decay function

$$E(i) = e^{\frac{-i}{B}}$$

This function has the convenient property that

$$\int_0^\infty E(i)di = B$$

meaning that as long as we can always add noise in accordance with our decay function, we will never overcharge the budget after leaking any number of logical bits. To convert between the continuous function $E(i)$ and a discounted entropy of our n-bit exit state, we calculate the height of a rectangle of base n with area equal to the area under $E(i)$ over the range of the n logical bits being leaked.

Back to the Future: Avoiding Impossible Time Quanta Once we know the desired amount of noise, Sandcastle must produce a probability distribution over the available exit states with the correct Shannon entropy. However, Sandcastle first attempts to reduce the entropy of the exit state space by removing times from the beginning of the list of time quanta if more than one is listed. This has the effect that a sandbox that returns before an early quantum may not actually have a chance to return at that time. While this seems like a simple waste of running time, it comes with the benefit that the noise Sandcastle must add to the return value can be reduced or eliminated, maximizing the chance we return the correct value to the caller. After all, most applications whose execution time is variable enough to necessitate listing multiple time quanta would rather receive the correct value slowly than an incorrect value quickly.

This transformation also avoids the issue that *we cannot go back in time*! If a sandbox returns in time for a 20 ms quantum, we cannot use an exit state with a 10 ms quantum in our noise distribution, because it is impossible for Sandcastle to actually honor that exit state; the sandbox has already run for longer than 10 ms. Worse, in high-noise situations, creating a distribution over exit states with the remaining possible times can fail to reach the necessary noise. Removing quanta first avoids this problem because it allows introducing noise arbitrarily close to the entropy of our reduced exit space. Thus, Sandcastle prevents timing side channels from leaking more entropy than noise can cancel out by *preventing them from happening altogether*.

Generating a Noise Distribution In cases where the desired noise exceeds what can be introduced by reducing the space of return times, Sandcastle creates a probability distribution over the return values to introduce the remaining noise. Each return value receives a rank, based on its Manhattan distance from its index to the index of the true return value. For example, a sandbox that lists return values [10, 11, 12, 13, 14] that returns the value 11 will produce the rank list [1, 0, 1, 2, 3]. We then convert the rank list to a probability distribution using the rule

$$\Pr[\text{Rank } r] = \delta^r \Pr[\text{Rank } 0]$$

for properly chosen exponential[3] decay factor δ, and $\Pr[\text{Rank } 0]$ determined by properly normalizing all the probabilities once δ is chosen. Then, we determine the correct value of δ via binary search[4] in order to achieve the entropy dictated by our noise aggregation function to generate the final probability distribution. Finally, we use our probability distribution to choose a cell corresponding to our noisy exit state using a secure random number generator.

Error Handling with Noise There is one final issue to discuss with respect to the noise model: error handling. So far, we have assumed that our sandboxes are *well-behaved*; that is, they will always complete before the longest available quanta and return a value that matches one of the return values in the list. If we relax this assumption, our noise model requires an additional layer of complexity.

Naively, we might treat these events as special, where errors are returned immediately without invoking the noise model at all, and design the noise model based on that errors never occurring. Unfortunately, this method introduces a side channel for the attacker, where she could convey sensitive information by deliberately returning an invalid value or timing out. Even with maximal noise added to the space of non-error exit states, there are still three basic exit states the attacker can observe: $\{$No Error, Timeout, Invalid Return Value$\}$. In other words, with this error handling method, it is fundamentally impossible to guarantee the sandbox leaks fewer than $\log_2 3$ bits.

Even with this lower bound, this naive error handling method is still the best for debugging and applications that do not strain the entropy budget. However, under this strategy it is not possible to make unlimited calls to high-entropy sandboxes. For these use cases, Sandcastle offers the option to either make errors fail silently with a random return value, or to treat errors as legitimate return values that can be occasionally introduced spuriously by the noise model. Both options avoid this lower bound.

5. Implementation

We implement Sandcastle as a set of libraries added to the Chromium browser, using a combination of C++ and Rust. All of our code is publicly available and integrates fully with Chromium's build system. We welcome the community to submit improvements to the performance and flexibility of Sandcastle itself, and especially encourage the development of JavaScript libraries that abstract over Sandcastle's core primitives to make safe, private web development simple and ergonomic for developers.

Sandbox We would like to reiterate that Sandcastle is not a JavaScript isolation framework on its own. Rather, it is a framework that can utilize existing and future isolation techniques for the web to enforce a robust and minimally invasive fingerprinting defense. The existing Web Workers [44] API is one such isolation method, which provides all of the necessary isolation guarantees in its current deployments

3. Like with the noise aggregation function, the use of an exponential decay function here is an arbitrary decision. If more flexibility is needed, other distribution shapes could be added in the future.

4. In principle, we could avoid this binary search if we could analytically determine the correct δ for a provided entropy. However, due to the number of edge cases in handling the edges of the list at all possible positions of the Rank 0 cell, we did not explore this direction.

across all major browsers. However, it incurs high overhead due to its use of dedicated threads for each task. Meanwhile, COWL [45] has demonstrated the ability to provide the same isolation guarantees at lower overhead.

Likewise, because Sandcastle's sandbox abstraction is the core primitive for developers to express what sensitive data particular code has access to, future deployments of Sandcastle would greatly benefit from a lightweight isolation framework, allowing our our browser API to spawn several sandboxes within a single web context with minimal overhead. Accordingly, the sandbox implementation we use for our evaluations in Section 7 is designed to have similar performance to COWL's Lightweight Workers [45].

Cashier and Noise Model Our implementation of the cashier and noise model incorporate all of the methods we describe in Sections 4.3 and 4.4, respectively. We also design and optimize both of these components for strong performance properties. Of particular note are our use of a combination of Tokio [46], a Rust concurrency library, and Chromium's internal JavaScript timing API to implement Sandcastle's time quantization without spawning additional threads, and our optimization of the noise model's rank and probability lists to require only constant memory space.

6. Case Studies

In this section, we provide a examples that illustrate how Sandcastle performs in the real world. Here, we demonstrate Sandcastle's abilities to both defend against fingerprinting and accommodate legitimate applications.

6.1. Fingerprinting Attack

Fingerprint.js [9] is a well-known open-source browser fingerprinting library, that offers TypeScript implementations that exploit several well-known fingerprinting vectors, including canvas fingerprinting, WebAudio signal analysis, hardware fingerprinting, and preference profiling, among others. In this section, we demonstrate how Sandcastle enables defending against fingerprinting attacks in a way that is much more flexible to the application than what is possible with currently deployed browsers. For each of the fingerprinting attacks in fingerprint.js that leverages a sensitive browser API, the sandboxing process looks the same, so we use WebAudio fingerprinting as an example.

The WebAudio library [47] is an expressive framework that enables the developer to choose and mix different audio sources, create audio visualizations, and apply a plethora of effects to customize the sound that plays in a web application. Unfortunately for the browser, this framework also provides the capability to retroactively analyze the sound played played in the audio channel using the `getChannelData()` function. Similar to canvas fingerprinting, this function creates a dangerous feedback loop between the application and the underlying hardware that enables the attacker to learn identifying information.

Example 3: WebAudio fingerprinting attack from fingerprint.js, modified to write the fingerprint to the DOM.

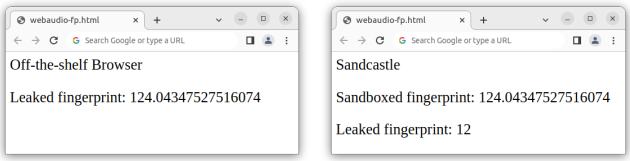
```
document.getElementById("leaked").textContent
  += getHash(buffer.getChannelData(0)
    .subarray(hashFromIndex));
```

Example 4: Rendering a full-entropy and leaked low-entropy version of the WebAudio fingerprint with Sandcastle.

```
sandcastle.createDOMSandbox(
  "opaque_1", "sb-frame", "parent", true,
  "<p id='sbfp'>Sandboxed fingerprint: </p>"
);
let leakedFP = sandcastle.sandbox(4, [],
  "opaque_1", function(hfi) {
    function getHash() { /* Code here */ }
    document.getElementById("sbfp").textContent
      += getHash(buffer.getChannelData(0)
        .subarray(hfi));
  }, hashFromIndex);
document.getElementById("leaked")
  .textContent += leakedFP;
```

Fingerprint.js exploits this feedback loop by playing a fixed audio track and hashing the audio channel data to obtain a fingerprint. Example 3 shows the critical section of code that performs the attack, which we modified to draw the hash directly to the DOM. Figure 3a shows the outcome of running the attack in unmodified Chrome, where the sensitive data leaked by the `getChannelData()` call is displayed as a hash on the (unsandboxed) DOM, demonstrating that the fingerprint can be leaked to the network, stored in untracked browser storage, or otherwise escape the client-side web context.

Contrast this with the variant of the same attack in Example 4. In this instance, we first create a DOM sandbox with Sandcastle's built-in `createDOMSandbox()` function, belonging to the opaque origin `opaque_1`. Next, we create a standard sandbox to place around the line of code that calls into the sensitive `getChannelData()` API. Supposing the developer decided they wanted to leak a secret-dependent value with a 4-bit granularity, the sandbox declares it will leak a 4-bit return value and that it will take ownership of the opaque origin `opaque_1` so that it has permission to write to the DOM sandbox we created. We also move the `getHash()` function declared elsewhere in the script inside the sandbox, enabling us to perform



(a) (b)

Figure 3: Sandcastle enables access to the entire WebAudio fingerprint only inside the sandbox.

the hash computation in the sandboxed context. Figure 3b shows the result of running Example 4 in Sandcastle, where the script writes two copies of the fingerprint to the DOM: one full-entropy copy inside the DOM sandbox, and a low-entropy copy that was leaked and charged by the cashier.

For completeness, we provide documentation of all of the types of fingerprinting the fingerprint.js library performs and their relationship to Sandcastle. Most of the fingerprints the library uses require browser APIs that must be invoked by a script. In addition to WebAudio channel data, these types of fingerprints include canvas fingerprinting as well as reading the color depth, CPU class, amount of system memory, font preferences, concurrency support, platform, screen frame, screen resolution, time zone, number of supported touch points, vendor, and vendor flavors.

Each of these fingerprints can be sandboxed to provide flexibility at the client-side in the same manner as in our WebAudio example, and it is simple to mark their APIs as sensitive so that the browser can charge their entire entropy if called outside the sandbox, as Privacy Budget does [3], [4], or in the strictest of defenses, block their use entirely outside the sandbox. Two other sources, the availability of session storage and the deprecated WebSQL library, while still sandboxable in this manner, would be prohibitively restrictive to block outside the sandbox. However, these features do not carry any information that could not be extrapolated from passive sources, particularly the user's browser version, so they are of little additional concern.

We also identify two classes of fingerprints that fingerprint.js uses that we cited in Section 8 as posing a challenge for Sandcastle. One attack performs explicit extension fingerprinting to determine the presence of Ad blockers, and another sequence of attacks read CSS display features including color gamut, contrast preference, dynamic range, forced colors, inverted colors, monochrome preference, and reduced motion preference.

6.2. Accommodating a Legitimate Application

Keeping with WebAudio as an example, we now demonstrate how Sandcastle can safely allow a legitimate application that would otherwise be fingerprintable. The WebAudio DJ [48] is a browser application that uses WebAudio to mix several pairs of percussion tracks fetched from the internet. Crucially, the app also takes the channel data generated by WebAudio and renders it to the DOM using a `Spectrogram`, provided by the third-party NexusUI [49] library. Since this rendering process introduces the same hardware feedback loop as in our previous example by using WebAudio's `AnalyzerNode` class, existing defenses would not allow this application.

Meanwhile, Sandcastle enables the DJ to run under a fingerprinting defense with minimal refactoring. First, we create create two DOM sandboxes that house the `<div>` elements the spectrogram uses to render the channel visualization. Then, we create a single large code sandbox in same opaque origin that contains all of the audio logic, including the `DeckPlayer` class, and the helper functions that are

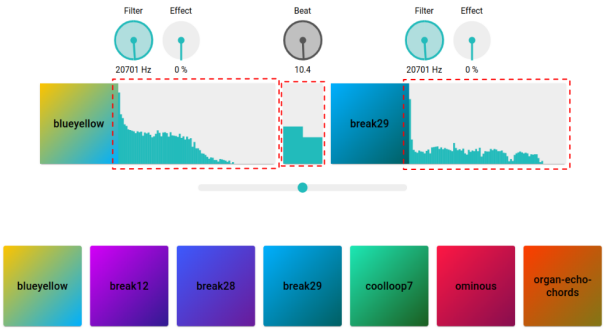


Figure 4: How Sandcastle accommodates the WebAudio DJ application. The red dotted lines are DOM sandboxes.

called by the event handlers. Then, each time the application receives an event that the user loads a new track, changes the volume, or moves one of the dials, we create a new sandbox in the event handler using the same opaque origin (sharing the context with the main sandbox), and calling the respective helper function from inside the sandbox. This design allows the spectrogram visualization to run entirely inside the sandbox, where Sandcastle can guarantee it poses no fingerprinting threat. Figure 4 shows a representation of the WebAudio DJ application, with the sandboxed DOM elements highlighted. Since the sandboxes in this example are purely taking in trusted data and rendering content inside DOM sandboxes, they do not leak any entropy[5], meaning this application can continue to respond to user requests indefinitely. To illustrate a case where Sandcastle's leaking mechanism would be useful, suppose we modified the DJ app to introduce an error reporting feature, where the application reports to the server that an audio track it served was broken or incompatible with the user's browser. To implement this, the developer could write the sandbox to return a boolean value representing whether WebAudio crashed when playing the track. Thus, only a single bit of entropy is leaked, and the browser is sure that the server only learns whether the track crashed while playing, not arbitrary channel data from the player.

7. Evaluation

In this section, we provide empirical data demonstrating Sandcastle's performance characteristics and its ability to reduce noise for leaked information. All of our experiments

5. We can use the *silent* error handling mode here to prevent the sandbox leaking a single bit by way of not returning a timeout each time it is called. If we do not do this, Sandcastle would eventually fall back to failing errors silently when the budget is exhausted, making it benign in this case. However, silent mode could be used to maintain our budget if other parts of the application happened to need it.

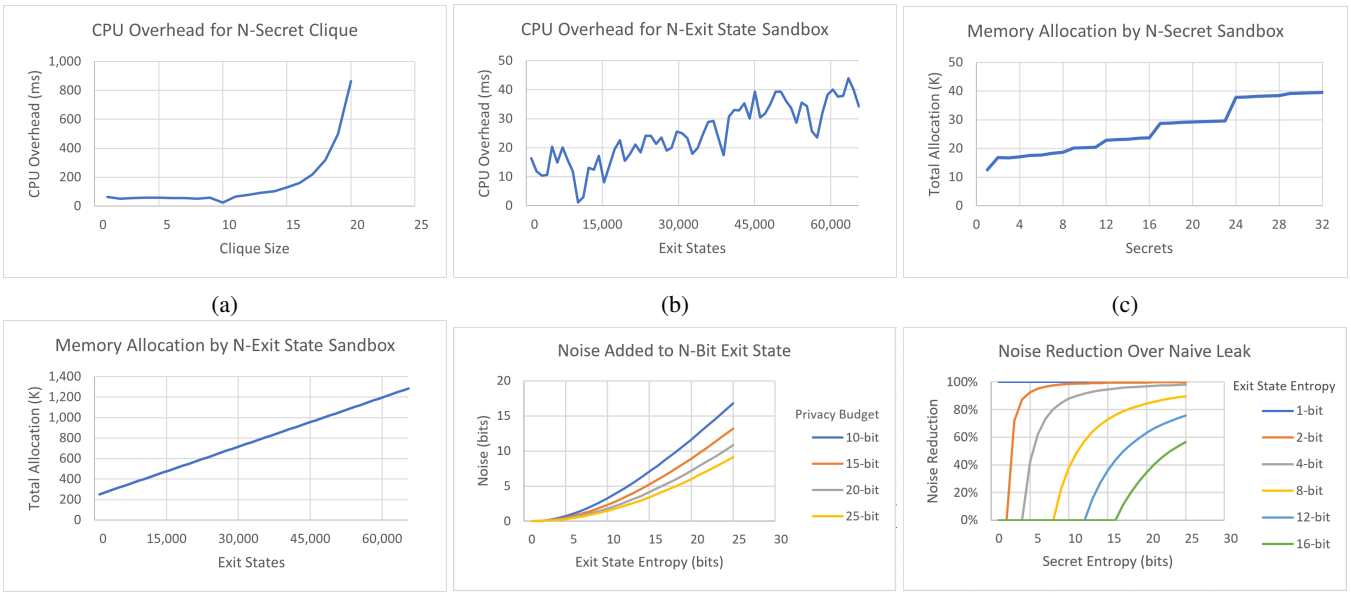


Figure 5: Our experimental results demonstrate that legitimate applications will incur minimal performance overhead from Sandcastle, and that reducing the entropy of a leak using the sandbox API achieves significant noise reduction.

were performed on an HP Elitebook 850 notebook with an Intel Core i5 CPU with 8 logical cores running at 1.7 GHz and 16G of DRAM, running Ubuntu Linux 20.04 LTS.

7.1. Performance

Computation Overhead We measure Sandcastle's computation time overhead across a variety of usage patterns using the Linux perf tool [50], comparing against the baseline of running identical sandboxed code in the calling context. For a correct comparison, we ignore any time spent sleeping to perform time quantization, since that time is controlled by the programmer, and measure only the overhead introduced by Sandcastle itself. To ensure a fair measurement across several amounts of noise, our times are averaged over five consecutive sandboxes run under the same cashier. Further, since the index of the actual return value in the provided list can affect the shape of the noise distribution, in our experiments that vary the number of exit states, we further average over five return indices at the 0th, 25th, 50th, 75th, and 100th percentiles of the list length.

First, we analyze the effect large cliques of secrets have on the performance of the cashier. As we explained in Section 4.3, the clique size is determined by the number of distinct browser secrets accessed by a single sandbox. While this clique size should be small for most applications, use cases such as 3D games and the system monitor application we described in section 3.3 will involve accessing a large number of these secrets at once. Figure 5a shows the computation overhead of running a sandbox accessing cliques of various sizes, with the number of exit states held constant at five. Our results from this experiment demonstrate that for cliques of size 13 or smaller, the computation overhead is roughly constant at 60-70 ms each time a sandbox exits. With cliques of size 14 and above, the computation time for the cashier to determine if entropy had been spent on the exponentially increasing number of subsets of the clique begins to dominate the computation time, with the overhead nearing 1 second at the largest clique size we measured, 20.

While our results indicate the potential for performance issues in applications that access a large number of secrets, it is important to observe that, under a reasonable budget, accessing that many secrets at once in an application like a game engine or a system monitor should never allow the cashier to prove a tighter bound for the cost than the sandbox's exit state entropy, since little to no information needs to leave the client in these use cases. This observation and the results of our experiment suggest that these types of applications would benefit performance-wise from an option to tell Sandcastle to disable the cashier's secret analysis and base the cost solely on the exit state entropy. This optimization will allow these applications to run with only the constant overhead we measured for small cliques.

Next, we examine the effect of increasing the entropy of the exit state affects the computation overhead by repeating the previous experiment, except we fix the clique size at 2 and vary the size of the sandbox's exit state space. Figure 5b shows our results, which indicate only a marginal increase in computation overhead by increasing the number of exit states. In fact, even at the largest return space we measured, 65536 exit states, corresponding to a 16-bit leak, Sandcastle incurs under 40 ms of computation overhead. Based on these results, we are confident applications that perform a single large data dump to a server will execute with minimal delay.

Memory Allocation Chrome's memory profiler is deprecated [51] and incompatible with current releases, so we evaluate Sandcastle's memory overhead using a standalone build that runs independently of the browser using a test harness, along with valgrind's memcheck tool [52], [53].

Interestingly, the large cliques our results in Figure 5a show can cause significant CPU overhead have minimal effect on memory consumption, because repeating the same clique several times does not require the cashier to store any new information each time. In fact, the analogous worst case for memory consumption, leaking every combination of secrets from a set of given size, is of little concern because it requires exponentially many sandboxes to reach. Nonetheless, it is desirable to understand how accessing more secrets affects a sandbox's memory overhead. Our results in Figure 5c indicate that each additional secret adds roughly 1K of allocation, which will produce minimal impact for applications like game engines and system monitors that operate a single sandbox that accesses many secrets.

Increasing the entropy of the exit state has a slightly less trivial impact on memory allocation, as we show in Figure 5d. Our results show that a sandbox with 256 exit states, leaking 8 bits of entropy, results in about 256K of memory allocation, which increases to about 1.28M for a 16-bit leak. However, the linear trend indicates most of the overhead comes from the compulsory allocation of memory for the exit states within Sandcastle, when setting up the sandbox, demonstrating that our optimizations to the noise model's rank and probability lists prevent any nontrivial memory usage in use cases that perform a large leak.

7.2. Noise Reduction and the Entropy Budget

Considering there are roughly 4 billion ($\sim 2^{32}$) internet users today, in order to provide reasonable anonymity, the browser should aim for a starting entropy budget no higher than 25-27 bits. Unfortunately, this budget must also cover passive fingerprinting surfaces that are outside of the browser's control once the page loads [4]. In fact, considering the plethora of passive fingerprinting sources studied in previous work [7], [8], [10], we suspect a starting budget for active fingerprints over about 20 bits would be unsound. Chrome's Privacy Budget proposal suggests replacing passive fingerprinting sources with active sources that can avoid being automatically charged for every web page [4], but because many of these sources are fundamental components of web protocols, we are pessimistic this effort will see much progress in the near future.

Moreover, a lower starting budget increases Sandcastle's ability to reduce noise over existing budget defenses. Of course, Sandcastle allows applications whose uses of sensitive data are entirely confined to the client side, such as a game rendering engine and the system monitor application we suggest in section 3.3, to run without any noise at all. On the other hand, applications like video ABR algorithms that do need to send some data to the server also achieve significant noise reduction over existing entropy budget solutions.

Figure 5e shows the amount of noise Sandcastle adds to exit states of varying sizes, assuming an previously uncharged cashier, for four different values for the budget available for active fingerprints. As expected, lower budgets result in more noise, and the difference in noise from varying the budget increases with the size of the exit state. The more critical observation, though, is that the average slope of the graph in Figure 5e between any two fixed exit state entropy values e_1 and e_2 increases as the budget decreases. This is important because the change in noise between e_1 and e_2 represents the noise reduction that Sandcastle provides for an e_1-bit exit state over naively leaking e_2 bits, which is all existing solutions allow. To our knowledge, Sandcastle is the first budget defense to formally define a noise model, but if we suppose an existing budget defense adopted a noise model using Sandcastle's noise aggregation function, the difference between e_1 and e_2 represents Sandcastle's noise savings. Our results in Figure 5e show that this savings for any given e_1 and e_2 increases as the starting budget decreases.

Figure 5f provides an alternate visualization of the data from Figure 5e, in the case of a 20-bit starting budget, that more clearly illustrates the noise reduction Sandcastle can achieve over existing budget defenses. Unsurprisingly, the noise reduction is highest when the exit state entropy remains low, with the 1-, 2-, and 4-bit exit states eliminating nearly 100% of the noise for sufficiently high-entropy secrets. Meanwhile, even the 16-bit exit state can achieve 56% noise reduction for a 25-bit secret entropy. In our video ABR example, 16 bits of exit state entropy corresponds to about 30 seconds of video for an ABR scheme with 4 choices of bit rate that runs every 3-4 seconds, indicating a large potential for video streaming apps, particularly short video services like TikTok and YouTube Shorts, to achieve significant noise savings. More generally, our noise reduction results demonstrate that Sandcastle achieves our goal of a *pay for what you leak* policy, allowing developers to eliminate most, if not all, of the noise from the data their applications leak.

8. Limitations

Scope of Defense Sandcastle reduces the entropy cost of invoking active fingerprinting sources when less or none of the entropy is necessary to send in a network request, particularly in the case when they are accessed via web scripts. It does not attempt to mitigate the entropy cost of passive fingerprinting sources that are fundamental components of HTTP requests. Likewise, as we mentioned in Section 2, Sandcastle considers extension fingerprinting out of scope, since it does not rely on any browser APIs. Further, browsers that employ existing blocking or budget defenses will require web developers to modify their applications to introduce sandboxes to obtain the client-side flexibility we highlight in this paper, and debugging such applications will require reasoning about the entropy the sandboxes access and leak in order to avoid overcharging the budget.

In addition, Sandcastle's protection is limited to features that are exposed by script APIs. While this is the case for most sensitive features applications may be interested in, a small number of CSS color, font, and display preferences are exposed via pure CSS, enabling script-free attacks such as text reflow and preference profiling in libraries like NoJSFingerprint [54]. Sandcastle does protect against these vulnerabilities for elements placed inside DOM sandboxes and when leaking the values through JavaScript, but it cannot prevent exploits that occur in unsandboxed CSS, such as notifying a server of the value of a preference through a conditional resource request. Many legitimate applications also use these features in CSS, making them nontrivial to update with sandboxing should the features be marked sensitive. As a result, we would like to see Sandcastle deployed alongside other defenses that mitigate the cost of these information sources that cannot be sandboxed.

Size, Scope, and Lifetime of Entropy Budget Though Sandcastle uses an entropy budget as its primary measurement of privacy, it does not take responsibility for setting the starting entropy budget. To choose a starting budget, browser developers should consider the level of anonymity they wish to provide and the identifiability of passive fingerprinting sources in their browser. We provide concrete data on how a browser's choice of the starting budget affects Sandcastle, particularly its noise model, in Section 7.2.

Sandcastle, like other entropy budget strategies, must take care to ensure the budget is sufficiently scoped to prevent fingerprinters from combining sensitive data from multiple budgets into a fingerprint. Within a single page, the CORS APIs [55] that enable communication between resources from different origins necessitate a budget lifetime at least as long as the lifetime of sessionStorage, also known as a *site navigation session* or *page session* [56]. This can be confusing to developers, as this allows external resources to consume part of their parent's budget, when such resources are otherwise isolated from their parent.

Furthermore, origins with pages that utilize longer-term storage features such as cookies and HTML5 storage must contend with budgets that persist for the lifetime of that storage, potentially spanning multiple pages. Because these scoping issues complicate application development, future work should explore methods for the application to explicitly delegate entropy to external resources and manage the lifetime of storage features.

9. Future Work

Our contributions in this paper raise several compelling questions about how we think about defenses against fingerprinting attacks. Here, we highlight two future directions we believe will be of particular interest to the community.

Robust Starting Budget In Section 7.2, we conjectured that a privacy budget above 20 bits for active fingerprints would be unsound. However, this was an estimate based on our analysis of the passive fingerprinting surfaces known to the academic community at this time. Designing a systematic method to determine, perhaps formally, the identifiability of the passive fingerprinting surfaces web browsers expose would improve browser developers' confidence in how to configure Sandcastle to provide its privacy guarantees.

Another related direction worth exploring is the impact that known limitations of k-anonymity have on the safety of particular choices of the starting budget. Narayanan and Shmatikov showed in 2008 [57] that k-anonymity offers weak protection for relatively sparse datasets. In the fingerprinting case, this means that an advertising network observing a small enough quantity of users could leverage normally non-identifiable fingerprints to pinpoint specific users with high accuracy. Likewise, choosing the starting budget based on real-world measurements of the number of users advertising networks observe could improve the robustness of entropy-based fingerprinting defenses like Sandcastle.

Automation An early design we considered for Sandcastle was to modify the browser's JavaScript runtime to perform secret taint-tracking at a program variable level. Though we abandoned this design on the grounds that it could not provide any timing channel defenses, it would have had the additional advantage of requiring no assistance from the developer to enforce its privacy guarantees. Meanwhile, there is potential for the sandbox mechanism to be similarly automated without dropping our timing channel defenses.

The main challenge is that the browser must know the space of possible exit states without the programmer specifying it ahead of time. For the timing, the browser could fix a reasonable default execution time, which Sandcastle already does if no return times are specified, or adopt a more advanced strategy based on previous execution and the level of a function's JIT optimization. However, there is no equivalent strategy for the return values. For those, the most likely direction is to perform static analysis of the code, either at the browser level or at the compiler level with less dynamic web languages like TypeScript. Either way, automated sandboxing would greatly improve Sandcastle's scalability and take a major step toward making fingerprinting defense the default across the web.

10. Conclusion

In this paper, we described the design and implementation of Sandcastle, a browser fingerprinting defense that provides web developers an expressive toolkit to prove to the browser their applications access identifiable information in a way that does not violate privacy. This ability enables the browser to permit the use of more of the innovative and exotic feature set the modern web ecosystem offers while mitigating the affects of these features on privacy. We showed, experimentally, that many applications leveraging these features will achieve significant noise reduction over the previous state-of-the-art.

Sandcastle's ability to isolate sensitive information from network access enables a brand new space of future web APIs that were not previously safe to implement at all, such as those that report detailed information about the hardware.

References

[1] "Do not track," https://www.eff.org/issues/do-not-track, 2022.

[2] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. New York, NY, USA: Association for Computing Machinery, 2014, p. 674–689. [Online]. Available: https://doi.org/10.1145/2660267.2660347

[3] A. White, "Privacy budget," https://developer.chrome.com/docs/privacy-sandbox/privacy-budget, 2020.

[4] M. West, "Privacy budget," https://github.com/mikewest/privacy-budget, Mar 2022.

[5] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, 2012, pp. 413–427.

[6] J. Angwin, "Meet the online tracking device that is virtually impossible to block," https://www.propublica.org/article/meet-the-online-tracking-device-that-is-virtually-impossible-to-block, July 2014.

[7] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," in *In proceedings of the 6th Workshop on Web 2.0 Security and Privacy (W2SP '12)*, M. Fredrikson, Ed. IEEE Computer Society, May 2012.

[8] N. M. Al-Fannah and W. Li, "Not all browsers are created equal: Comparing web browser fingerprintability," in *Proceedings of the 12th International Workshop on Security (IWSEC '17)*, 2017.

[9] "Fingerprint.js quick start guide," https://dev.fingerprint.com/docs, 2022.

[10] J. V. Monaco, "Device fingerprinting with peripheral timestamps," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*, 2022, pp. 1018–1033.

[11] "Firefox's protection against fingerprinting," https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting.

[12] B. P. Team, "Fingerprinting defenses 2.0," https://brave.com/privacy-updates/4-fingerprinting-defenses-2.0/.

[13] "Browser fingerprinting," https://support.torproject.org/glossary/browser-fingerprinting/.

[14] "What are grey borders resized on tor browser window?" https://support.torproject.org/tbb/maximized-torbrowser-window/.

[15] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *Proceedings of 42nd IEEE Symposium on Security and Privacy (S&P '21)*, 2021, pp. 1143–1161.

[16] "Fingerprint randomization," https://brave.com/privacy-updates/3-fingerprint-randomization/, 2020.

[17] "Brave, fingerprinting, and privacy budgets," https://brave.com/privacy-updates/3-privacy-budgets/, 2019.

[18] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 820–830. [Online]. Available: https://doi.org/10.1145/2736277.2741090

[19] P. Laperdrix, B. Baudry, and V. Mishra, "Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques," in *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS '17)*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 97–114.

[20] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Extension breakdown: Security analysis of browsers extension resources control policies," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*. USA: USENIX Association, 2017, p. 679–694.

[21] P. Picazo-Sanchez, A. Sjösten, S. Acker, and A. Sabelfeld, "Latex gloves: Protecting browser extensions from probing and revelation attacks," in *Proceedings of the 27th Network and Distributed Systems Security Symposium (NDSS '19)*, 02 2019.

[22] S. Karami, P. Ilia, K. Solomos, and J. Polakis, "Carnus: Exploring the privacy threats of browser extension fingerprinting," in *Proceedings of the 28th Network and Distributed Systems Security Symposium (NDSS '20)*, 02 2020.

[23] O. Starov and N. Nikiforakis, "Xhound: Quantifying the fingerprintability of browser extensions," in *Proceedings of the 38th IEEE Symposium on Security on Privacy (S&P '17)*, 05 2017.

[24] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickel, P. Ilia, Y. Shoshitaishvili, A. Doupé, and J. Polakis, "Unleash the simulacrum: Shifting browser realities for robust extension-fingerprinting prevention," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*, 2022.

[25] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, "Everyone is different: Client-side diversification for defending against extension fingerprinting," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, 2019.

[26] "At home testing," https://www.ets.org/gre/revised_general/register/at_home.

[27] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, "Learning in situ: A randomized experiment in video streaming," in *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI '20)*. USA: USENIX Association, 2020, p. 495–512.

[28] A. O. El Meligy, M. S. Hassan, and T. Landolsi, "A buffer-based rate adaptation approach for video streaming over http," in *Proceedings of the 19th Wireless Telecommunications Symposium (WTS '20)*, 2020, pp. 1–5.

[29] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *In Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. New York, NY, USA: Association for Computing Machinery, 2017, p. 197–210. [Online]. Available: https://doi.org/10.1145/3098822.3098843

[30] H. Yeo, Y. Jung, J. Kim, J. Shin, and D. Han, "Neural adaptive content-aware internet video delivery," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. USA: USENIX Association, 2018, p. 645–661.

[31] J. Kim, Y. Jung, H. Yeo, J. Ye, and D. Han, "Neural-enhanced live streaming: Improving live video ingest via online learning," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. New York, NY, USA: Association for Computing Machinery, 2020, p. 107–125. [Online]. Available: https://doi.org/10.1145/3387514.3405856

[32] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," *Commun. ACM*, vol. 56, no. 3, p. 91–99, mar 2013. [Online]. Available: https://doi.org/10.1145/2428556.2428577

[33] "The structured clone algorithm," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[34] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL '19)*, vol. 3, no. POPL, Jan 2019. [Online]. Available: https://doi.org/10.1145/3290390

[35] "Flexible box layout," https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout.

[36] "Lazy loading," https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_Loading.

[37] "Anonymous iframe," https://wicg.github.io/anonymous-iframe/#explainer.

[38] W. W. I. C. Group, "Anonymous iframe," https://github.com/WICG/anonymous-iframe, June 2022.

[39] "Cross-origin embedder policy," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy.

[40] "iframe: The inline frame element," https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox.

[41] "Origin," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin.

[42] R. M. Gray, *Entropy and Information Theory*, 2nd ed. New York: Springer, 2014.

[43] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[44] "Web workers api," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[45] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining javascript with cowl," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USA: USENIX Association, 2014, p. 131–146.

[46] "Tokio - an asynchronous rust runtime," https://www.tokio.rs.

[47] "Web audio api," https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API.

[48] "Web audio dj," https://googlechromelabs.github.io/web-audio-samples/demos/dj/.

[49] B. Taylor, "Nexus web audio interfaces," https://nexus-js.github.io/ui/.

[50] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, 2022.

[51] "Deep memory profiler," https://www.chromium.org/developers/deep-memory-profiler, 2015.

[52] "About valgrind," https://valgrind.org/info/about.html, 2022.

[53] "4. memcheck: a memory error detector," https://valgrind.org/docs/manual/mc-manual.html/manual.html, 2022.

[54] "No-javascript fingerprinting," https://noscriptfingerprint.com, 2022.

[55] "Cross-origin resource sharing," https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[56] "Window.sessionstorage," https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage.

[57] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P '08)*, 2008, pp. 111–125.

Appendix A. Formal Definition of the Cashier

In Section 4.3, we gave a high-level introduction to how we use the cashier to optimally charge secrets in the presence of uncertainty of the behavior of sandboxes. In this appendix, we formally define the behavior of the cashier, discuss the limitations that arise from the cashier's uncertainty, and, most importantly, show that the algorithm we use for the cashier in Sandcastle determines the optimal safe cost for a set of secrets returned by a sandbox.

Recall we began Section 4.3 with an informal explanation of the *cashier uncertainty principle*, which posits that a sandbox that operates on multiple secrets cannot safely make any assumptions about how much of the exit state's entropy is attributed to each secret, beyond that no secret can leak more entropy than it originally contains. Now, we formally express this phenomenon.

Proposition A.1 (Cashier Uncertainty Principle)**.** *Let A and B be secrets that begin with E_A and E_B bits of entropy, respectively. Let S be a sandbox that leaks L_A bits worth of A and L_B bits worth of B. Consider an imperfect cashier C that instead subtracts $L_A - \epsilon$ bits from A's remaining entropy, and $L_B + \epsilon$ bits from B's remaining entropy when S returns, where ϵ is a small positive number. Assuming $L_B + \epsilon \leq E_B$, there exists another sandbox S' that allows an attacker to leak more combined entropy from S and S' than C charges for them.*

Proof. To determine an S' that allows the attacker to force C to undercharge her, first observe that C overcharged B by ϵ bits when S returned. Now, suppose we let S' leak all of B, and access no other secrets. S' will then be charged all of the remaining uncharged bits of B, or $E_B - (L_B + \epsilon)$ bits (since we assumed $L_B + \epsilon \leq E_B$, this quantity cannot be negative; in other words, C did not subtract more than E_B bits from B's entropy when S returns). Meanwhile, we also know that the entropy C charged for S is $(L_A - \epsilon) + (L_B + \epsilon) = L_A + L_B$. Thus, C will charge S and S' a total of $(L_A + L_B) + (E_B - (L_B + \epsilon)) = L_A + E_B - \epsilon$ bits. However, in reality, S leaked L_A + L_B bits, and since S leaked L_B bits of B, that means S' will leak $E_B - L_B$ bits. In total, S and S' leak $(L_A + L_B) + (E_B - L_B) = L_A + E_B$ bits. Thus, C undercharged S and S' collectively by ϵ bits! Thus, we found an S' that causes C to undercharge S and S' in aggregate. □

As we explained in Section 4.3, Sandcastle copes with the cashier uncertainty principle by making the cashier expressive enough to reason about uncertainty about the secrets responsible for leaked entropy through its technique of set saturation. Next, we formally define saturation:

Definition A.1. *Let C be a cashier. A set of secrets S, whose members have E bits of entropy in total, is **saturated** under C if C has charged at least E bits to the privacy budget for sandboxes that only operate on members of S.*

In other words, a set of secrets is saturated when it has been charged for all the entropy its members contain. Further, recall again from Section 4.3 that saturation also applies to sets that have not been explicitly charged by the cashier, and it is often useful to consider these sets. Consider the following theorem.

Proposition A.2. *Let C be a cashier, and S be a set of secrets that is saturated under C. Any future sandbox operating on only members of S can safely leak any data without charging any additional entropy to the privacy budget.*

Proof. The proof of this statement is quite simple. Suppose the members of S have E bits of entropy in total. By the definition of saturation, this means C has already charged at least E bits of entropy to sandboxes that operate only on S or any of its proper subsets. Thus, the first E bits that

were charged from these sandboxes must not correspond to any sandboxes that leak any information for secrets outside of S. Since the members of S have only E bits of entropy in total, it is impossible to undercharge any subset of S in a future sandbox, even if it is charged no entropy, since the entire entropy of S has already been charged. □

Proposition A.2 says nothing about whether any sandbox has actually accessed all of S at once. In Section 4.3, we gave an example where the sets $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$ were all halfway saturated, but those half-saturations all added together to fully saturate $\{A, B, C\}$, meaning a set like $\{A\}$ could return for free from then on, even though neither of $\{A, B\}$, $\{A, C\}$ was saturated.

This observation leads to another intriguing question: if the minimum safe cost for a set of secrets can be upper bounded by sets that have not been explicitly charged, how do we *find* them, and even if we do, how do we know what that bound is? Let us address the second part of this question first. Consider the following extension to our earlier definition of saturation:

Definition A.2. *Let C be a cashier and S be a set of secrets, whose members have E bits of entropy in total. The **saturation distance** of S under C is the number of bits C must further charge sandboxes that operate only on S (or its proper subsets) before S is saturated.*

In fact, combining this definition with our earlier definition of saturation gives the saturation distance of S as

$$\Delta(C, S) = \text{Ent}(S) - \text{Ch}(C, S)$$

where $\text{Ent}(S)$ is the total entropy of all secrets in S, and $\text{Ch}(C, S)$ is the number of bits C has charged for S and its proper subsets. Now, we show the following more general version of Proposition A.2:

Proposition A.3. *Let C be a cashier, and S be a set of secrets. Let T be a subset of S. It is always safe for C to charge T a number of bits equivalent to S's saturation distance.*

Proof. We already derived $\Delta(C, S) = \text{Ent}(S) - \text{Ch}(C, S)$, which implies C has charged sandboxes operating only on elements of S a total of $\text{Ent}(S) - \Delta(C, S)$ bits. So, if C charges T $\Delta(C, S)$ bits, the total bits charged to sandboxes operating on S or its subsets would total $\text{Ent}(S)$. Since we never need to charge more than $\text{Ent}(S)$ bits for a set of sandboxes operating only on S or its subsets, this is sufficient. □

Our goal, then, for a sandbox operating on T, is to find the superset of T that can guarantee the lowest safe cost for T, or, equivalently, has the smallest saturation distance. We will touch on how to *find* this set soon, but first, it is useful to state that this method actually achieves the optimal cost.

Proposition A.4. *Consider a sandbox operating under a cashier C and returning excess entropy of a set of secrets T. Let P be the set containing T and all of its strict supersets. The minimum safe cost C can charge for T is* $\min_{S \in P} \Delta(C, S)$.

Proof. Let S^* be the optimal S above that has the smallest saturation distance. Suppose by way of contradiction that it is safe for C to charge T $\Delta(C, S^*) - \epsilon$ bits, where ϵ is a small positive number. Also observe that it must be that $\Delta(C, S^*) \leq \text{Ent}(T)$, because if $\Delta(C, S^*) > \text{Ent}(T)$, then $\text{Ent}(T)$ would be a trivial safe cost for T, violating our assumption that S^* is the optimal element of P. Now, since $\Delta(C, S^*) \leq \text{Ent}(T)$, this means that the attacker has spent at least $\text{Ent}(S^* \setminus T)$ bits on S^*, and in the worst case, we have to assume the attacker used them to obtain all of $S^* \setminus T$. Now, since we are supposing the sandbox in question allows her to also obtain T for $\Delta(C, S^*) - \epsilon$ bits, that means she obtains $(S^* \setminus T) \cup T = S^*$ for a total of $\text{Ch}(C, S^*) + \Delta(C, S^*) - \epsilon = \text{Ent}(S^*) - \epsilon$ bits. Thus, the attacker is undercharged for S^* by ϵ bits!

Now, it is important to notice that this does not *necessarily* indicate a leak. Instead, one of the following two events could have occurred:

1) There exists a subset of S^*, call it S_{sub}, that guarantees a tighter bound for T.
2) There exists another set of secrets, call it S', that contains elements outside of S^* and guarantees that some of S^* has been charged other than those that just operate on elements of S^*.

We will now show that neither of these events can actually happen under these conditions.

First, consider S_{sub}. Suppose for sake of contradiction that we knew a tighter bound for how much of S_{sub} has been leaked than what is trivially known just by using S^*. In other words, $\text{Ch}(C, S_{sub}) > \text{Ch}(C, S^*) - \text{Ent}(S^* \setminus S_{sub})$. Next, notice this quantity cannot be negative because $\text{Ch}(C, S^*) < \text{Ent}(S^* \setminus S_{sub})$ would imply

$$\text{Ch}(C, S^*) < \text{Ent}(S^* \setminus S_{sub})$$

$$\text{Ent}(S^*) - \Delta(C, S^*) < \text{Ent}(S^* \setminus S_{sub})$$

$$\text{Ent}(S_{sub}) < \Delta(C, S^*)$$

since $\text{Ent}(S_{sub}) + \text{Ent}(S^* \setminus S_{sub}) = \text{Ent}(S^*)$. However, this means $\text{Ent}(S_{sub})$ would again place a trivial bound on the cost of T, violating our assumption that S^* is optimal. Now we know that $\text{Ch}(C, S^*) - \text{Ent}(S^* \setminus S_{sub})$, observe that $\text{Ch}(C, S_{sub}) > \text{Ch}(C, S^*) - \text{Ent}(S^* \setminus S_{sub})$ would imply

$$\Delta(C, S_{sub}) = \text{Ent}(S_{sub}) - \text{Ch}(C, S_{sub})$$

$$< \text{Ent}(S_{sub}) - \text{Ch}(C, S^*) + \text{Ent}(S^* \setminus S_{sub})$$

$$= \text{Ent}(S^*) - \text{Ch}(C, S^*)$$

$$= \Delta(C, S^*)$$

Thus, $\Delta(C, S_{sub}) < \Delta(C, S^*)$. Now, since S_{sub} must contain all elements of T (otherwise it achieves a bound no tighter than $S_{sub} \cup T$, which would then be subject to the same constraints), S_{sub} must also be in P. This means

$\Delta(C, S_{sub}) < \Delta(C, S^*)$ is a contradiction to our assumption S^* has the smallest saturation distance in P.

Second, suppose for sake of contradiction that S' exists. Let $S_C = S^* \cup S'$. In order for S_C to guarantee some elements of S^* have been charged, it must be that $\text{Ch}(C, S') > \text{Ent}(S' \setminus S^*)$. However, since $S' \setminus S^* = S_C \setminus S^*$, this implies

$$\text{Ch}(C, S') + \text{Ch}(C, S^*) > \text{Ch}(C, S^*) + \text{Ent}(S_C \setminus S^*)$$

But since $\text{Ch}(C, S') + \text{Ch}(C, S^*) = \text{Ch}(C, S_C)$,

$$\text{Ch}(C, S_C) > \text{Ch}(C, S^*) + \text{Ent}(S_C \setminus S^*)$$

Next, because Ent(S_C) - Ent($S_C \setminus S^*$) = Ent(S^*),

$$\text{Ch}(C, S_C) - \text{Ent}(S_C) > \text{Ch}(C, S^*) - \text{Ent}(S^*)$$

Multiplying through by -1 gives

$$\text{Ent}(S_C) - \text{Ch}(C, S_C) < \text{Ent}(S^*) - \text{Ch}(C, S^*)$$

which, by definition of saturation distance, gives

$$\Delta(C, S_C) < \Delta(C, S^*)$$

Now, S_C must be in P, because S_C is a superset of S^*, so it certainly must be a superset of T, since S^* is itself a superset of T. Thus, S_C turns out to be an element of P with a smaller saturation distance than S^*, which is a contradiction.

Finally, since we showed that both items 1 and 2 led to a contradiction, this implies charging T $\Delta(C, S^*) - \epsilon$ bits allows the attacker to leak bits the cashier is unaware of. □

Moving ahead, we next address how to efficiently find the set of secrets that guarantees the lowest safe cost for T. Since checking every superset of T would be incredibly inefficient, we will instead formalize the relationship between secrets as a problem of graph connectivity, enabling us to trim the search space. Consider a universe U of all secrets accessed by sandboxes under a single entropy budget. We now provide the following useful definitions:

Definition A.3. *Let C be a cashier and U_C (the universe of C) be the set of all secrets accessed by sandboxes that have operated under C. The **secret graph** of C is the undirected graph containing each secret in U_C as a vertex, and an edge connecting each pair of secrets (S_1, S_2) where at least one sandbox operating under C has accessed both S_1 and S_2.*

Definition A.4. *Let C be a cashier, U_C be the universe of C, and G be the secret graph of C. Let T be a subset of U_C. Now, consider the set of vertices V_T of G corresponding to the secrets in T. Let V_{N_T} be the set of vertices reachable in G from at least one element of V_T. The **neighborhood** of T under C is defined as the set of secrets in U_C corresponding to the vertices in V_{N_T}.*

With this definition, and based on our discussion so far, it is natural to suspect the following statement is true:

Proposition A.5. *Let C be a cashier, U_C be the universe of C, and T be a subset of U_C. Let P be the set of all sets containing all elements of T and zero or more other elements of U_C. Let δ^* be the minimum saturation distance under C in P, or $\min_{P_i \in P} \Delta(C, P_i)$. There exists at least one set in P, P^*, such that $\Delta(C, P^*) = \delta^*$ and P^* contains only elements in the neighborhood of T under C.*

In other words, we should never need to look outside the neighborhood of T to find a set of secrets that guarantees the lowest safe cost for T. This fact turns out to be true, but we can actually prove something even stronger!

Imagine we had a cashier who had previously charged sandboxes for $\{A, B\}$, $\{B, C\}$, $\{C, D\}$, and $\{E, F\}$, and we then have a fifth sandbox leak A alone. Which sets should we consider? Proposition A.5 tells us that any sets containing E or F are useless, since they are not in the neighborhood of A, but sets like $\{A, B, D\}$ would still be in play. However, notice that this set *breaks the chain* between D and A; in other words, D is only useful to A because it places a lower bound on the amount the cashier has charged C. However, because C was excluded from the set, the cashier will not consider that relationship when calculating the saturation distance of $\{A, B, D\}$. Thus, sets that break connectivity like this are completely useless! We can begin to formalize this using the following lemma:

Lemma 1. *Let G be a secret graph with at least two connected components. Let G_S and G_T be two such components in G, with S and T the respective corresponding sets of secrets. Let C be a cashier that calculates saturation distance by considering only previous exit states containing only secrets in G. Then $\Delta(C, S) \leq \Delta(C, S \cup T)$.*

Proof. Since G_S and G_T are not connected in G, there will be no edges between any element in G_S and any element in G_T, and consequently no previous exit states C will consider when calculating $\Delta(C, S \cup T)$ that will not be considered for either $\Delta(C, S)$ or $\Delta(C, T)$. Thus,

$$\text{Ch}(C, S \cup T) = \text{Ch}(C, S) + \text{Ch}(C, T)$$

Since S and T are disjoint,

$$\text{Ent}(S \cup T) = \text{Ent}(S) + \text{Ent}(T)$$

Subtracting the first identity from the second gives

$$\text{Ent}(S \cup T) - \text{Ch}(C, S \cup T) = (\text{Ent}(S) - \text{Ch}(C, S)) + (\text{Ent}(T) - \text{Ch}(C, T))$$

Which by the definition of saturation distance, implies

$$\Delta(C, S \cup T) = \Delta(C, S) + \Delta(C, T)$$

Finally, since $\Delta(C, T) \geq 0$, this implies

$$\Delta(C, S) \leq \Delta(C, S \cup T)$$

□

This lemma is useful in combination with Proposition A.4, which indicates the relationship between saturation distance and the minimum safe cost for a set of secrets. Using these facts in conjunction allows us to formalize the broken chain phenomenon as the *Secret Graph Theorem*:

Proposition A.6 (Secret Graph Theorem)**.** *Let C be a cashier, U_C be the universe of C, and T be a subset of U_C. Let P be the set of all sets containing all elements of T and zero or more other elements of U_C. Let δ^* be the minimum saturation distance under C in P, or $\min_{P_i \in P} \Delta(C, P_i)$. Let G be the secret graph of C. There exists at least one set in P, P^*, such that $\Delta(C, P^*) = \delta^*$, and the induced subgraph of G comprised of the vertices corresponding to the elements of P^* contains at least one vertex corresponding to an element of T in each of its connected components.*

Proof. Suppose this were false. Then, any and every possible P^* would contain some subset of its elements whose corresponding vertices are unreachable from any of the vertices corresponding to elements of T in the induced subgraph of G corresponding to P^*. (To simplify our further language, we will refer to secrets and their corresponding vertices in G as synonymous when discussing reachability and connectivity).

Consider any such P^*. Let G^* be the corresponding induced subgraph of G containing the elements of P^*. Let C^* be the subset of P^* whose elements are reachable in G^* from at least one element of T, and D^* be the unreachable part, equivalent to $P^* \setminus C^*$. Notice that both of these sets must contain at least one element, since C^* contains all of T itself, and an empty D^* would violate our assumption that part of P^* is not reachable from any of T in G^*.

Now, consider any one connected component of D^*, call it D'. Now, when C considers P^* as a potentially optimal superset of T to minimize T's cost, C will find by Lemma 1 that

$$\Delta(C, C^*) \leq \Delta(C, C^* \cup D')$$

This fact rules out that D' by itself is useful, but not necessarily all of D^* at once[6]. To resolve this case, consider all connected components of D^*, $\{D_1, D_2, D_3, ..., D_n\}$. Imagine what would happen if C had further charged exit states depending on $\{D_1, D_2\}$, $\{D_2, D_3\}$, ..., $\{D_{n-1}, D_n\}$ $\frac{\epsilon}{n-1}$ bits each for some positive ϵ. This would connect all components of D^* in G^*. Call this new connected component D^{**}. Since D^{**} is now a single connected component, Lemma 1 would apply between C^* and D^{**}, implying

$$\Delta(C, C^*) \leq \Delta(C, C^* \cup D^{**})$$

Now, it follows from the definition of saturation distance that C charging more exit states can never *increase* the saturation distance of C^*. Specifically, this is because no charge C makes can ever decrease the total amount charged to C or any of its subsets. Thus, $\Delta(C, C^*)$ is monotonically decreasing as the number of charges C makes increases over time. Combining this fact with the observation that the phantom exit states used to create D^{**} could only decrease $C^* \cup D^*$'s saturation distance, this means

$$\Delta(C, C^*) \leq \Delta(C, C^* \cup D^*)$$

6. We do not need to consider *subsets* of D^* other than D' because such a subset would act as D^* in a different secret set checked by C (assuming it performed its search using the rule posed by Proposition A.5), meaning this entire argument applies to that set as well.

However, this would make C^* another possible P^*, but since every element of C^* is connected in G^* to an element of T, this contradicts our assumption that every possible P^* contains a component unreachable in G^* from any element of T. □

Thus, we have shown that a cashier C wanting to bound the cost for an exit state depending on a set of secrets S needs only to consider supersets S_{sup} of S such that the induced subgraph of C's secret graph corresponding to S_{sup} has every vertex reachable from at least one vertex corresponding to an element of S. Combining Propositions A.4 and A.6 tells us that this strategy will find the optimal safe cost for S. Using these principles, we formally define the cashier algorithm in Algorithm 1.

Algorithm 1 Determine cost for secret set S

```
G ← Secret graph of C
N ← Vertices in G reachable from S, excluding S itself.
OPT ← ∞
for C ∈ combinations(N) do
    G' ←Induced subgraph of G containing S and C
    valid ← True
    for v ∈ vertices(G') do   ▷ Reachability requirement
        for s ∈ S do
            if v reachable from s in G' then
                r ← True
            end if
        end for
        if not r then
            valid ← False
        end if
    end for
    if valid then
        U ← 0                          ▷ Entropy spent under G'
        for T ∈ subsets(vertices(G')) do
            U ← U+ total C has charged to exactly T
        end for
        if U > OPT then
            OPT ← U
        end if
    end if
end for
    return OPT
```

Our optimizations reduce the time complexity of the cashier from exponential in the number of secrets the cashier has seen to exponential in the size of the largest clique in the secret graph. However, this is still not a polynomial time algorithm due to the exponential number of supersets to check. However, we believe this is tolerable because the number of secrets actually accessed by a practical website should be relatively small, and even an adversarial developer aiming to make the cashier run slowly would only be hurting their own performance. Also, the cashier can approximate the solution in these degenerate cases by short-circuiting after checking a certain number of supersets, which is guaranteed to uphold soundness by only possibly overestimating the optimal cost.