Formal Verification of Browser Fingerprinting and Mitigation with Inlined Reference Monitors

Nathan Joslin(✉), Phu H. Phung, and Luan Viet Nguyen

Department of Computer Science, University of Dayton, Dayton, OH 45469, USA
{joslinn1,phu,lnguyen1}@udayton.edu

Abstract. Browser fingerprinting is a technique that identifies user devices by exploiting differences in software and hardware configurations. This technique is used in both benign applications, such as multi-factor authentication, and malicious ones, like web tracking and the disclosure of private information. Existing work has proposed various defense mechanisms against malicious browser fingerprinting and evaluated them using empirical experiments and analysis. While this approach demonstrates the effectiveness of mitigation methods, it does not provide proof of reliability. As browser fingerprinting research continues to advance and gain popularity across the web, there is an increasing need to verify the safety of user data and the reliability of protection mechanisms. In this paper, we develop formal models of both a browser fingerprinting tool and a controller capable of enforcing fingerprinting mitigation techniques. Specifically, we model an Inlined Reference Monitor for a canvas fingerprinter that intercepts JavaScript function calls and provides runtime policy enforcement. Our framework is highly extensible, allowing it to model a wide range of fingerprinting strategies and defenses. Using Computation Tree Logic, we formally define safety and liveness properties, demonstrating that our model successfully enforces key anti-fingerprinting techniques, such as randomization and API blocking.

Keywords: Formal Verification · Formalization · Browser Fingerprinting · Privacy · Security · Mitigation Techniques · UPPAAL

1 Introduction

In response to growing concerns over data privacy, many countries have begun updating legislation to meet the requirements of the digital age, with the European Union's General Data Protection Regulation (GDPR) and the ePrivacy directive being particularly influential [6,38]. These regulations require web pages to be more transparent on how and why they collect user data. The effect of these regulations is frequently seen in the form of consent pop-up windows for browser cookies [38]. The combination of increasing transparency and efforts put forth by regulatory bodies has led many users to decline the usage of browser

L. Horn Iwaya et al. (Eds.): NordSec 2024, LNCS 15396, pp. 303–321, 2025.
https://doi.org/10.1007/978-3-031-79007-2_16

cookies, which has driven cookie-based web trackers to explore alternatives, such as browser fingerprinting. Recent studies estimated that over 25% of the top 1,000 websites and nearly 10% of the top 100,000 use browser fingerprinting in some forms [17,35]. Browser fingerprinting presents significant challenges for web users seeking greater control over their digital footprint [40]. As a stateless mechanism, unlike cookies, browser fingerprints can be collected by web pages, allowing for the tracking of user activity without their knowledge or consent [20]. This method is particularly concerning because it even affects users who may not be uniquely identifiable, leaving them vulnerable to attacks that exploit software with known vulnerabilities [9]. Additionally, browser fingerprinting can lead to the exposure of sensitive information [19], the re-spawning of cookies after deletion [14], and even the identification of users through the linkage of fingerprints to social media accounts [20]. When used for involuntary tracking or unauthorized data collection, such practices clearly violate user privacy and can be deemed malicious.

However, it is important to recognize that browser fingerprinting also serves a protective role in certain contexts. For instance, it is increasingly used as an additional layer in multi-factor authentication schemes, particularly on login pages, where it helps safeguard users from fraud and phishing attacks [9,24, 35]. According to the United States Federal Trade Commission, this particular application has become increasingly important in recent years as online fraud and phishing attacks continue to grow, significantly contributing to the overall 14% increase in fraud between 2022 and 2023 [12]. Fingerprinting's utility in these scenarios is evident, as it is commonly employed on login and sign-up pages to enhance security and prevent unauthorized access [35]. Thus, while browser fingerprinting poses significant risks to user privacy, it also plays a crucial role in enhancing online security. The challenge lies in balancing these competing interests—ensuring fingerprinting is used responsibly to protect users without infringing on their privacy rights.

As a stateless tracking mechanism, it is challenging for average users not only to protect themselves from malicious actors but also to allow fingerprinting from trusted sources. Although some tools exist, e.g., fingerprint randomization in the Brave browser [36], there is still much work needed to increase the transparency, effectiveness, and, most importantly, reliability of anti-fingerprinting technologies. The first step to gaining control over browser fingerprinters is detection, as demonstrated in many studies with different proposed methods [1,11,17,18]. The second step is the design and empirical evaluation of mitigation methods, many of which have already been proven to be effective [2,7,20,25]. With extensive research conducted on these initial two steps, particularly in recent years, we are now well-positioned to shift our focus toward the next step: formalizing browser fingerprinting and its mitigation methods. This step allows researchers and developers alike to know the exact limitations and effectiveness of both fingerprinters and the defenses against them. A few works have proposed formal definitions of browser fingerprinting [4,21]. However, no prior work has built

formal models or used formal methods to mathematically prove or verify the reliability of fingerprinters or the defenses against them.

In this paper, we develop formal models for a fingerprinting system and a corresponding defense mechanism using the formal modeling and verification framework UPPAAL [27]. Our approach involves defining three components: a fingerprinter, a server, and a controller, which together form a network of timed automata operating in parallel during the verification process [3]. The controller adopts the Inlined Reference Monitor (IRM) approach [10], which models the interception of functions executed by the fingerprinter at runtime. The IRM approach is particularly versatile, supporting the enforcement of several well-established fingerprinting mitigation strategies, including randomization, which introduces variability to fingerprinting data; normalization, which standardizes outputs to reduce uniqueness; and API blocking, which prevents unauthorized access to critical browser functions. We have designed our models to be extensible, providing a foundational framework that can be easily adapted to model more sophisticated fingerprinting techniques or advanced defense mechanisms. This extensibility is crucial, as it allows for the integration of future developments in both fingerprinting tactics and corresponding countermeasures.

We evaluate our model's effectiveness in enforcing these mitigation techniques by abstracting them into formally defined safety and liveness properties using Computation Tree Logic (CTL) [34]. Finally, through our formal evaluation using tools like UPPAAL, we aim to demonstrate the practical viability of our approach. Our results suggest that it is indeed possible to selectively allow or deny fingerprinting by domain as well as through more fine-grained policy enforcement depended on runtime factors; such as the methods used to fingerprint or how often the fingerprint is collected. This capability is particularly significant because it addresses an often overlooked aspect of existing anti-fingerprinting research: the need to balance security with usability. By enabling fine-grained control over which domains can perform fingerprinting, our model offers a more nuanced and practical solution to the challenges of browser fingerprinting in real-world applications. This work lays the groundwork for future research, which could explore more sophisticated fingerprinting techniques, further enhance the adaptability of the framework, and bridge the gap between theoretical models and practical implementation in web security. The models are available on Github[1].

2 Background and Related Work

2.1 Browser Fingerprinting

Modern web browsers provide significant information to the web pages they render, such as their configurations or the software running them. Individually, these pieces of information are often referred to as attributes and provide great value to the web pages that use them. Attributes as simple as a device's screen

[1] https://github.com/nathan-joslin/BrowserFingerprintingFormalization.

size allow web pages to properly format themselves to provide the best user experience for as many devices as possible. This aggregation of browser data is known as browser fingerprinting and has been proven to be able to uniquely identify browsers [9,20,24] and even track activity across the web [28,32,39,40].

Browser fingerprinters target attributes based on their potential entropy, or information gain, in an effort to yield a collision-resistant identifier. A fingerprint of one browser will differ from another as a result of variations in software and hardware configuration, such as installed fonts or the user agent [9,24]. More complex fingerprinters will introduce data-generating fingerprinting methods. Instead of simply collecting attributes, these methods generate unique data by running scripts[2] that exploit particular browser APIs. One of the most studied data-generating methods is canvas fingerprinting. Due to its prevalence in research, proven effectiveness [24,29], active use in both public and commercial libraries [37], and increased complexity compared to other methods, we choose to model a canvas fingerprinter to evaluate the effectiveness of mitigation methods. An example of a canvas fingerprint and the results of a simple mitigation method can be seen in Fig. 1.

(a) Original (b) Poisoned

Fig. 1. An example of a canvas fingerprint generated with amiunique.org [23]. The second image was subject to a simple implementation of a mitigation method known as canvas poisoning, which draws hashes of the current date and time randomly on the element.

2.2 Browser Fingerprinting Mitigation Approaches

Related work has demonstrated a variety of anti-fingerprinting mechanisms, which can be organized into three basic groups: normalization, randomization, and blocking [7]. First, the normalization technique takes a "hide in the crowd" approach. Also known as *attribute standardizing*, this method aims to reduce the entropy of fingerprints by setting attributes to a value shared among a sufficiently sized user base [7]. It is important to note that even minor deviations from the normalized user base can make users stand out more than they typically would without the technique. Not only has this method been shown to be effective [2,7], but it is also actively used by the Tor Browser [7,15]. The second well-known mitigation method is randomization. Also known as *attribute varying*, this method aims to create a "moving target" [7]. This technique will regularly change browser attributes or introduce noise to those that are generated in order

[2] Data generating fingerprinting scripts are often referred to as *challenges*.

to change the collected fingerprint. Random alterations to the data each time a fingerprint is collected make it increasingly difficult to track a user over time or across the web. Related works have shown that randomization is one of the most effective methods [24,26]. Finally, we have blocking techniques. Also known as *interaction blocking*, these methods block the execution of a particular API or interactions with particular third-party domains [7]. While some techniques block entire APIs, such as Tor blocking the canvas API [15], other techniques use partial or temporal API blocking [17]. This technique can be particularly aggressive, leading to frequent major and minor website breakage [17].

Many works have demonstrated the effectiveness of normalization, randomization, and blocking methods through empirical evaluation. While the contributions of these works are extremely valuable, they are limited in their ability to evaluate the reliability of these mitigation methods. As far as we know, few works have used formal methods in the domain of browser fingerprinting. In particular, two works have proposed formal definitions for browser fingerprinters with respect to their properties, such as uniqueness, robustness, or stability [4,21]. However, no work has used formal modeling and verification techniques to mathematically validate that defenses against fingerprinters behave as their empirical analyses show in all possible scenarios. The purpose of this work is to complement the aforementioned related works that perform empirical analyses by applying formal methods to the same mitigation techniques they propose.

2.3 Inlined Reference Monitors

Inlined Reference Monitors (IRMs) are a robust security mechanism that embeds runtime policy enforcement directly into applications, ensuring that execution adheres to predefined security policies [10]. By integrating policy checks within the code, IRMs can effectively regulate and modify behavior during runtime, making them particularly suitable for controlling JavaScript execution in web environments [31]. They operate by intercepting targeted JavaScript functions and enforcing actions as dictated by the specified security policies [16]. The versatility of IRMs extends beyond simple function interception. In more complex scenarios, it is necessary to monitor not only the initial function call but also the behavior of objects returned by these functions. This layered monitoring capability allows IRMs to adapt to dynamic and evolving threats by providing fine-grained control over both the inputs and outputs of function calls. The context-aware nature of IRMs—gained through runtime interception enables the enforcement of nuanced, fine-grained security policies that can account for the specific parameters and conditions of each function call [13].

These capabilities make IRMs highly effective in implementing well-known browser fingerprinting mitigation strategies such as normalization, randomization, or API blocking. In the cases of normalization or randomization, such a controller is able to modify the output of the function it intercepts, thereby standardizing or adding noise to the output value. In cases of API blocking, the controller is able to return an empty or nil value to prohibit the caller from further working with the output value, perhaps even throwing an error. Their

ability to enforce all three mitigation methods highlights their suitability as a core component of our proposed controller model.

3 System Modeling

3.1 UPPAAL and Timed Automata

UPPAAL is a tool for automatically verifying system requirements specified as temporal logic, such as CTL. It provides a framework for modeling, simulating, and verifying real-time systems, where the correctness of the system is essential for ensuring reliability and safety [8,27]. The system in question is represented as a network of finite-state machines extended with real-valued clocks, a formalism known as timed automata, which allows for the modeling of systems where timing constraints are critical [5,27]. Each component of a timed automaton is characterized by a set of locations (representing states), an initial location, location invariants (conditions that must hold as long as the system remains in a particular location), a set of clocks (real-valued variables that progress uniformly), a set of actions (events that trigger transitions), and edges (transitions) between locations, each labeled with actions. These elements collectively define the dynamic behavior of the system, including how the system evolves over time and how it responds to various events.

In UPPAAL, the model is composed of one or more timed automata running in parallel, synchronized through channels or shared variables. This parallel composition enables the modeling of complex systems, where multiple components operate concurrently while adhering to strict timing constraints. The global state of the system is determined by the combined states of all automata and the values of global variables, with transitions occurring according to the rules defined by the automata and the constraints imposed by the clocks. In this work, we leverage UPPAAL as a framework for modeling and verifying a browser fingerprinting system along with its associated safety and liveness requirements, expressed as CTL formulas. In the following sections, we will provide a detailed presentation of the proposed formal model using UPPAAL.

3.2 System Overview

In this section, we provide a comprehensive overview of our system, which effectively models a fingerprinting attack and an Inlined Reference Monitor (IRM) enforcing defense mechanisms. The defined system has three main components: a **Fingerprinter**[3], a **Controller**, and a **Server**. Components communicate with one another using designated synchronization channels over state transitions. To model the **Controller** as an IRM, the **Fingerprinter** component is instrumented with a channel synchronization and corresponding state invariant for each monitored function while the **Controller** listens to those same channels in parallel. These channels are optionally instrumented with corresponding shared

[3] System components are emphasized in **bold**.

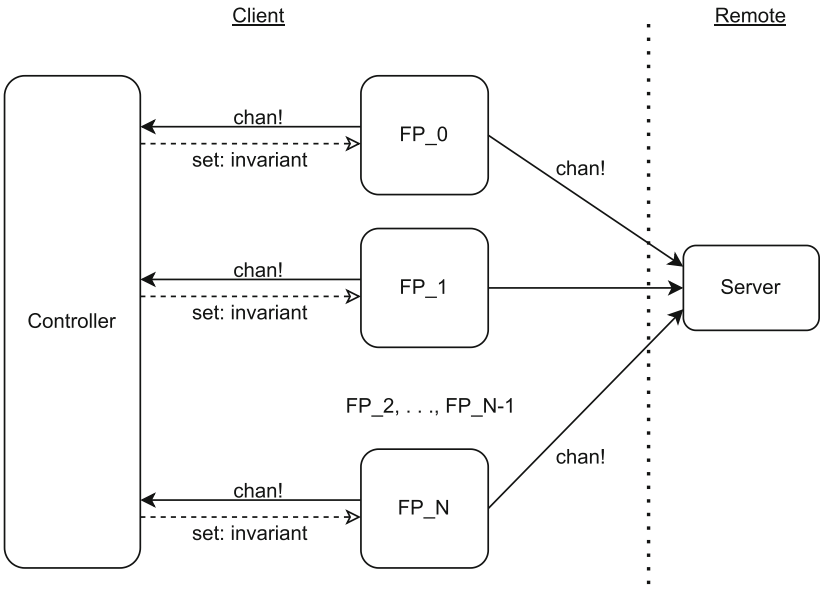


Fig. 2. Overview of communication between components. Unique synchronization channels, with corresponding state invariants, are defined for all functions monitored by the **Controller** for each **Fingerprinter**, with additional channels for submitting data to the **Server**. As such, for any system configuration, the total number of channels is defined by $f(x, y) = xy + y$; where x is the number of functions monitored by the **Controller** and y is the number of **Fingerprinter** components.

variables that represent the context around the function call, such as parameters. With this approach, the total number of active channels is determined by the abstracted functions the **Controller** monitors as well as how many **Fingerprinter** components are in the declared system. By defining a unique set of channels for each **Fingerprinter**, the system is able to model an IRM that's capable of enforcing policies on a per-component basis. Likewise, we define unique location invariants for the **Fingerprinter** to complement the unique set of channels. This completes the instrumentation of the **Fingerprinter** to support the IRM approach. As such, components may model a specific domain running a script or actions on a particular JavaScript object. This aspect of the design is crucial in giving the template the flexibility to support a wide variety of fingerprinting schemes as well as defense mechanisms. More information on how each component behaves during synchronizations on state transitions is described in detail in subsequent sections. The full implementation of our models is available on Github[4].

[4] https://github.com/nathan-joslin/BrowserFingerprintingFormalization.

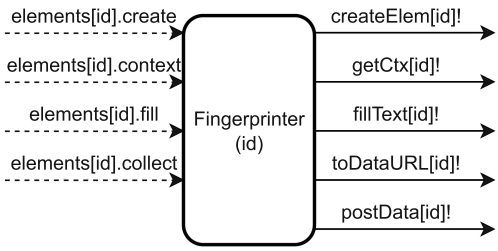
3.3 Modeling the Fingerprinter

The **Fingerprinter** component is an abstraction of a fingerprinting technique created through static analysis of multiple scripts to identify similarities in API usage. Due to the vast scope of browser fingerprinting, we focus on one of the more complex techniques - dynamic data generation. In particular, we abstract a well-known method that exploits the widely available canvas API [20,22,29]. Although we only provide a model of a canvas fingerprinter, analogous approaches may be used for other data-generating methods, such as `AudioContext` fingerprinting [33], or even more trivial methods that simply aggregate data rather than generate it [9,20,40]. By modeling one of the more complex methods while considering extensibility, our goal is to design a system that can serve as a template for modeling entire fingerprinting schemes with the ability to evaluate a broad range of fingerprinting defenses. At a minimum, the system has a single **Fingerprinter** component; however, it easily scales to handle multiple ones.

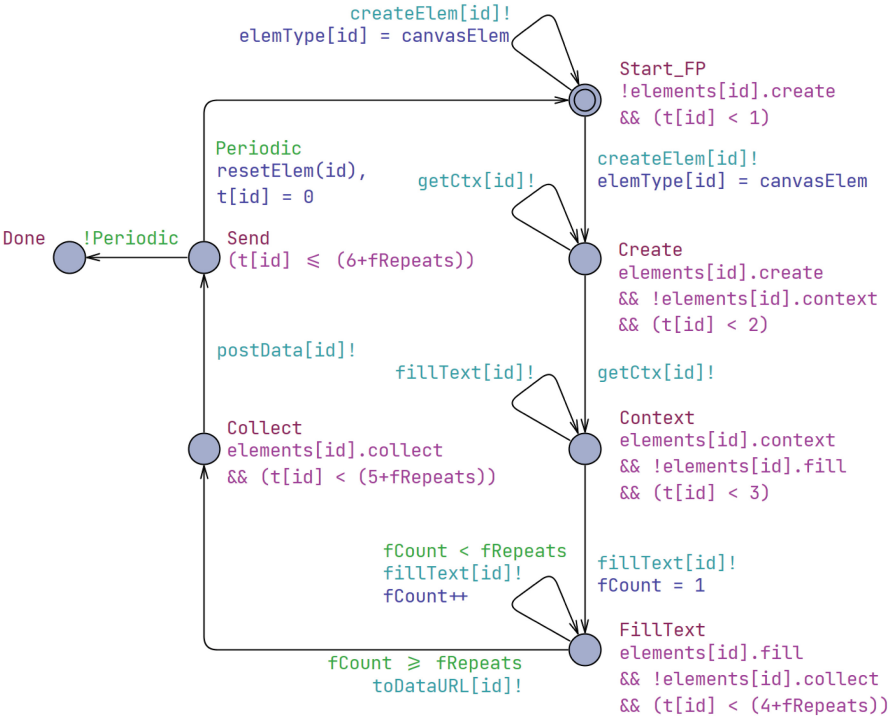
We analyzed two implementations of canvas fingerprinters to base our model on, one from a real-world application and the other from related research. The first is from the widely used and open-sourced browser fingerprinting library `fingerprintjs2` [37], which provides a wide variety of fingerprinting functions allowing developers to easily integrate browser fingerprinting methods into the functionality of their websites. The second is from related work done by Laperdrix et al. [22,24], who push the limits of what canvas fingerprinting is capable of through rigorous analysis and experimentation.

The **Fingerprinter** component, depicted in Fig. 3b, models the main steps taken by canvas fingerprinting algorithms. The steps include creating a canvas element, getting the canvas context, drawing on the context, collecting the fingerprint value, and finally sending the value to a database. Within our model, these main steps are represented by the following locations respectively: *Create*[5], *Context*, *FillText*, *Collect*, and *Send*. As the automata is an abstraction of a running script, it follows the edges leading into each state model function execution, with the states themselves modeling the successful completion of the function call. Most of the modeled functions are only executed once, with the exception of drawing on the canvas context, which is the expected behavior of a well-written script. The remaining locations and transitions are added instrumentation to allow a wider variety of simulations that may require singular or repetitive fingerprinting. This particular mechanism is useful for evaluating the behavior of fingerprinting defenses over time; such as mitigation through randomization, which aims to ensure the fingerprint value is different each run. As mentioned previously, the channel synchronizations on state transitions are added instrumentation to model the **Controller** as an IRM; which is discussed further in the next section. Finally, the majority of locations of the **Fingerprinter** include timing constraints to support the verification of liveness properties.

[5] Locations of system components are emphasized in *italics*.

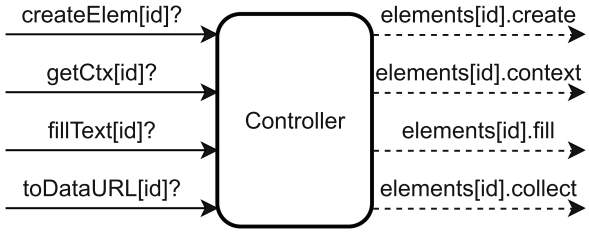


(a) **Fingerprinter Component**

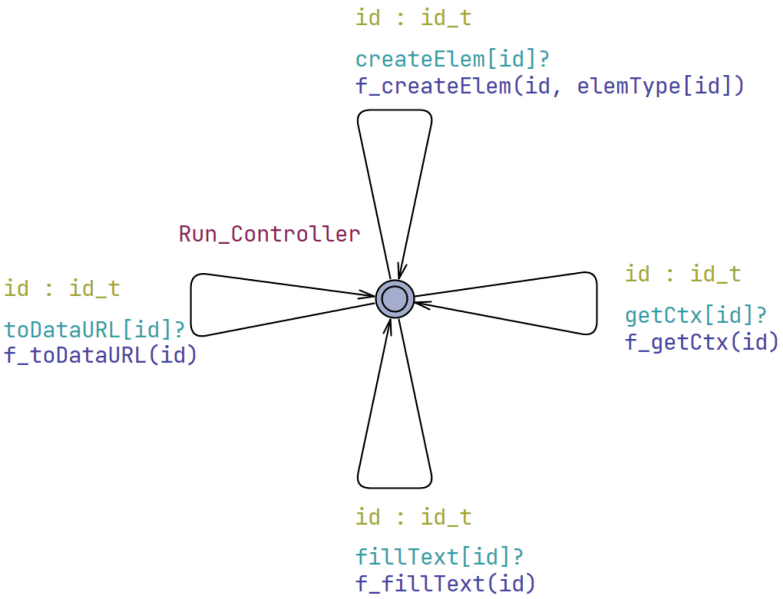


(b) **Fingerprinter Automata**

Fig. 3. The **Fingerprinter** models a running canvas fingerprinting script. The main transitions, see right and bottom, synchronize with the **Controller** via designated channels for each function. The invariants on transition destination states are managed by the **Controller**, which allows or denies particular actions based on the policy configuration. The remaining transitions, see left and top, post the data to the server and either terminate or reset the component.



(a) **Controller Component**



(b) **Controller Automata**

Fig. 4. The **Controller** models a running Inlined Reference Monitor actively intercepting the function calls it observes. Each transition is synchronized with a sending **Fingerprinter**, with the update functions performing the policy evaluation. Internally, the policy evaluation will set the appropriate sending **Fingerprinter** state invariant.

3.4 Modeling the Controller

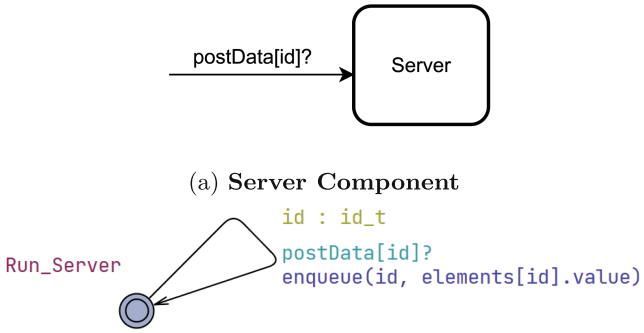
The **Controller** abstracts an IRM running in a client's browser. The system always has a single **Controller** as we aim to model the fingerprinting of one client. We find this to be sufficient as the purpose of this work is to formally

verify the defense mechanisms themselves, discussed in Sect. 2.2, not the effects they have on an entire fingerprinting scheme. Such an endeavor may require modeling hundreds of thousands, if not millions, of clients at the same time; which current formal verification tools are not equipped to handle. We refer readers to empirical analyses in related work for more information [7,25,26]. Furthermore, it follows that the behavior and capabilities of the same controller on different clients would be identical.

As previously mentioned, the **Fingerprinter** is instrumented with channel synchronizations and state invariants for each monitored function. This allows us to accurately model the behavior of an active IRM. The **Controller** listens to this set of channels, and upon receiving a channel synchronization, it evaluates some arbitrary set of policies that determine if the monitored function is allowed to be called. During this evaluation time, the **Controller** can take a variety of actions to support more fine-gained approaches other than simply allowing or blocking, such as calling the original function and then modifying the data returned by it before returning to the **Fingerprinter**. Ultimately, the **Controller** appropriately sets the **Fingerprinter's** corresponding state invariant, either allowing it to continue to the next state or preventing it. We note that based on the purpose of the **Controller** to immediately react to actions taken by the **Fingerprinter**s, as reflected by the synchronization channels and invariant controls, it follows that we do not instrument it with timing constraints. Finally, Table 1 displays the Controller Transition Update Functions. These functions are invoked when channel synchronizations are initiated by the **Fingerprinter** and received by the **Controller**. The update functions evaluate the relevant policies and set the appropriate invariants for the sending **Fingerprinter**.

Table 1. Controller Transition Update Functions. Update functions are called when the channel synchronizations are sent by a **Fingerprinter** and received by the **Controller**. The related policies are evaluated by the update functions, which ultimately set the invariants for the sending **Fingerprinter** appropriately.

Function	Meaning	Policies	Invariant
f_createElem	Create a DOM element	p_createElement	create
f_getCtx	Get canvas' drawing context	p_getCtx	context
f_fillText	Generate data by drawing on canvas context	p_fillText	fill
f_toDataURL	Collect fingerprint value	p_poison p_toDataURL	collect



(a) **Server Component**

(b) **Server Automata**

Fig. 5. The **Server** models a remote server receiving fingerprint values. Upon receiving a channel synchronization from a **Fingerprinter**, the **Server** will read and store the fingerprint data from a shared variable.

3.5 Modeling the Server

The **Server** abstracts a remote server and database as a singular component. Although separate server and database components would be a more accurate representation of a real-world system, we choose to combine the two as an evaluation of the remote subsystems is not within the scope of this paper. Similarly, we do not include support for **Server** responses to the **Fingerprinter**. In doing so, we still fulfill the purpose of this component, instrumenting our model to enable the evaluation of fingerprint attributes' values over time while reducing the state space of our model.

The **Server** component, depicted in Fig. 5, models the storing of fingerprint data in a remote location. The **Server** blocks until it receives a channel synchronization from any **Fingerprinter**, subsequently storing the **Fingerprinter**'s attribute value in an underlying database. To further reduce state space, the underlying database is equipped with a configurable capacity. Upon submission of fingerprint data when at capacity the **Server** will remove the oldest value before storing the new one.

4 Verification Regarding Unsafe Regions

In this section, we discuss and evaluate several requirements that a client may have when looking to defend against browser fingerprinting attacks. We will first discuss these requirements informally, then translate them into formal safety properties, and finally utilize UPPAAL's simulation and verification tools to evaluate our model. It is important to note that we have chosen the following system setup and configuration to illustrate the capabilities of our model; however, it can easily be extended to handle more components or more complex requirements.

Table 2. Informal System Requirements. The evaluated system's policy configuration expressed as a set of informal requirements.

Component	Requirement	Mitigation Method
FP_0	Allow fingerprints to be freely collected, without intervention.	None
FP_1	Allow fingerprints to be collected, but poison its data.	Randomization
FP_2	Do not allow fingerprints to be collected whatsoever.	API Blocking

4.1 Informal Requirements and Policy Configuration

The following requirements, shown in Table 2, aim to evaluate the capability of our model to enforce well-known browser fingerprinting mitigation methods, in particular randomization and API blocking. Note that although we do not model the normalization technique, the enforcement mechanism itself remains the same as randomization; however, instead of adding noise to the attribute value, it would simply be set to a standard one. It follows that an evaluation of our model's capability to enforce the randomization method transfers over to normalization. Note that we do not evaluate the necessary amount of noise to add when randomizing values or the proper value to spoof when normalizing. Such an endeavor is not possible with current formal verification tools due to the state space expansion that results from modeling a sufficiently sized user base. We refer the reader to empirical analyses from related works for more information [9,24, 29,40]. Furthermore, we evaluate our models ability to simultaneously mitigate attacks from some components while allowing it from others.

In Table 2, the first requirement does not mitigate at all, instead allowing the fingerprinting to occur as if the **Controller** was not there. This serves to demonstrate the system's ability to permit fingerprinters from one domain while simultaneously preventing it from others. The second requirement uses a randomization mitigation approach, which takes the form of adding noise to the fingerprint attribute data each time it is collected. As such the fingerprint attribute values between two different collection attempts should never be the same; or at minimum distinct enough to thwart a tracking system. The final requirement uses a blocking approach, which completely prevents the data collection from happening. The formal safety and liveness properties that verify these requirements are defined in Table 4, which will be introduced later.

4.2 Verifying Formal Safety and Liveness Properties

To formally evaluate all three requirements at once, we first directly translate the requirements into a well-defined policy configuration, which is set up to control each **Fingerprinter** in a separate fashion. The policy configuration is comprised of a set of lists, with individual lists configured as an `allowlist` or `blocklist`. We chose this approach to support extensions to our model that aim to evaluate more complex requirements. The resulting configuration, depicted in Table 3, instructs the controller to allow fingerprinting from **FP_0**, randomize the fingerprint of **FP_1**, and completely block the collection of data by **FP_2**. In

Table 3. Policy Configuration. The system's policy configuration adheres to the requirements in Table 2. Policies are implemented as allowlists or blocklists, and are evaluated by the **Controller** during function call interception. Table 1 outlines which policies are evaluated by which function calls.

Policy	Type	FP_0	FP_1	FP_2
p_createElem	blocklist	false	false	false
p_getCtx	blocklist	false	false	false
p_fillText	blocklist	false	false	false
p_toDataURL	blocklist	false	false	*true*
p_poison	allowlist	*true*	false	false

this configuration, all policies are allowed by default except for *p_poison*, which is blocked by default, reflecting the outlined requirements.

Now that the system is setup to enforce our defined requirements, we may begin our formal evaluation. We will perform our verification through the use of safety and liveness properties expressed using Computation Tree Logic (CTL), which are depicted in in Table 4. The first requirement states that component **FP_0** should be allowed to fingerprint the client. We construct safety property A that ensures the fingerprint attribute value of **FP_0** is *never* poisoned and is able to be collected. The second requirement states that component **FP_1** should be allowed to fingerprint the client but the fingerprint attribute value should be poisoned. We construct safety property B that ensures the fingerprint attribute value of **FP_1** is *always* poisoned and is able to be collected. Finally, the third requirement states that component **FP_2** should not be allowed to collect the client's fingerprint data. We construct safety properties C and D that ensure component **FP_2** is never able to collect the fingerprint attribute value; two methods of expressing the same property. By using UPPAAL's verifier, we confirm that our model satisfies all four of the aforementioned safety properties.

We then extend our safety evaluation with a set of reachability and liveness properties. Properties F, G, and H further verify the three requirements respectively by ensuring the components are able to reach the locations their intended to. It follows that property F complements A, G complements B, while property H complements properties C and D. As expected, property H is not satisfied since **FP_2** is blocked from collecting the fingerprint value, i.e. it cannot reach the *Collect* location. Finally, the liveness properties I and J ensure randomization is correctly applied. As expected, property J is not satisfied due to the enforcement of randomization.

Note that as we utilize UPPAAL to verify a network of timed automata, our approach inherits the verification complexity of UPPAAL when checking the system against Computation Tree Logic (CTL) requirements. The time complexity of UPPAAL's verification algorithm is exponential in the number of clocks and variables within the timed automata network, as it relies on symbolic model checking techniques [8,27].

Table 4. Safety properties (*A*-*E*) and liveness properties (*F*-*J*) that reflect the requirements outlined in Table 2. These properties are used to evaluate our system's ability to reliably enforce well-known fingerprinting mitigation methods, namely randomization and API blocking. The **Sat.** column indicates whether or not a property is satisfied. By design, some properties should not be satisfied to validate the expected effects of mitigation methods.

Prop.	Sat.	Format	Value
A	true	CTL	`A[] FP_0.Collect imply (elements[0].value > 0)`
		Meaning	For all reachable states, component **FP_0** being in the location *Collect* implies that its attribute value is not poisoned.
B	true	CTL	`A[] FP_1.Collect imply (elements[1].value < 0)`
		Meaning	For all reachable states, component **FP_1** being in the location *Collect* implies that its attribute value is poisoned.
C	true	CTL	`A[ ] FP_2.Collect imply evalPolicy(p_toDataURL, 2)`
		Meaning	For all reachable states, component **FP_2** being in the location *Collect* implies the policy configuration allows it, i.e. the attribute data is allowed to be collected.
D	true	CTL	`A[ ] !FP_2.Collect`
		Meaning	For all reachable states, component **FP_2** is never in the *Collect* location.
E	true	CTL	`A[ ] Server.db[2].len == 0`
		Meaning	For all reachable states, the server does not store any values for **FP_2**.
F	true	CTL	`E<> FP_0.Collect`
		Meaning	The *Collect* location is reachable in the **FP_0** component.
G	true	CTL	`E<> FP_1.Collect`
		Meaning	The *Collect* location is reachable in the **FP_1** component.
H	false	CTL	`E<> FP_2.Collect`
		Meaning	The *Collect* location is *not* reachable in the **FP_2** component.
I	true	CTL	`A<> ((Sever.db[0].len > 0) && (Server.db[0].entries[0] == Server.db[0].entries[1]) && (Server.db[0].entries[1] == Server.db[0].entries[2]))`
		Meaning	Eventually all database entries for **FP_0** are the same.
J	false	CTL	`A<> ((Server.db[1].len > 0) && (Server.db[1].entries[0] == Server.db[1].entries[1]) && (Server.db[1].entries[1] == Server.db[1].entries[2]))`
		Meaning	Eventually all database entries for **FP_1** are the same.

5 Conclusion and Future Work

In this paper, we modeled a canvas-based browser fingerprinter and an Inlined Reference Monitor (IRM) using timed automata. To evaluate our models, we first defined a set of requirements reflecting well-known fingerprinting mitiga-

tion methods. We then used Computation Tree Logic (CTL) to formally express these requirements as well-defined liveness and safety properties. Using the formal verification tool UPPAAL, we assessed our model's behavior against these properties. Our evaluation confirmed that the model effectively enforces several widely recognized mitigation strategies for browser fingerprinting, including randomization, normalization, and API blocking. This result demonstrates that policy enforcement through an IRM is a robust and reliable method for defending against browser fingerprinting attacks. Furthermore, we showed that fine-grained policies can selectively allow fingerprinting from trusted domains while preventing it from malicious ones.

In real-world applications, browser fingerprints are typically constructed using multiple attributes; however, our focus was exclusively on a single canvas-based attribute. Despite the inherent complexity of data-generating fingerprinters, we successfully designed a foundational model that also considers similar, more straightforward fingerprinting techniques. This allowed us to create and verify a flexible framework that is readily extensible to model more comprehensive fingerprinting methods, a task we have reserved for future work. Furthermore, as part of future work, we plan to develop a comprehensive formal model of a real-world browser fingerprinter which includes a variety of fingerprint attributes. Since a comprehensive formal model is beyond the scope of the current study, we also do not provide an attack model at this stage. Such an endeavor should address key questions regarding the minimal mitigation steps necessary to prevent fingerprinting attacks. For instance, it is crucial to determine how many fingerprint attributes need to be randomized to effectively protect the user, as well as to ensure that sensitive information does not leak to untrusted domains through unintended information flows, as explored in [30]. Finally, we see significant potential in automating the translation of a verified system into practical applications. Future research could involve developing code-generation scripts that produce JavaScript for web applications, thereby bridging the gap between anti-fingerprinting research and its real-world implementation.

Acknowledgements. This work received partial support from the Ohio Department of Higher Education (ODHE) through the Strategic Ohio Council for Higher Education (SOCHE) and Ohio Cyber Range Institute (OCRI) sub-awards and from the National Science Foundation (NSF) grant NSF-CRII-2245853. We would like to extend our appreciation to the anonymous reviewers for their valuable feedback.

Disclosure of Interests. The author have no competing interests to declare that are relevant to the content of this article.

References

1. Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., Preneel, B.: Fpdetective: dusting the web for fingerprinters. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security,CCS 2013, pp. 1129-1140. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516674
2. Ajay, V.L., Guptha, A.M.: A defense against javascript object-based fingerprinting and passive fingerprinting. In: 2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS), pp. 1–6. IEEE (2022)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
4. Andriamilanto, N., Allard, T., Le Guelvouit, G., Garel, A.: A large-scale empirical analysis of browser fingerprints properties for web authentication. ACM Trans. Web (TWEB) **16**(1), 1–62 (2021)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. Formal Methods Des. Real-Time Syst. 200–236 (2004)
6. Commission, E.: Principles of the gdpr. https://commission.europa.eu/law/law-topic/data-protection/reform/rules-business-and-organisations/principles-gdpr (2024), Accessed 29 Sep 2024
7. Datta, A., Lu, J., Tschantz, M.C.: Evaluating anti-fingerprinting privacy enhancing technologies. In: The World Wide Web Conference, pp. 351–362 (2019)
8. David, A., Du, D., Larsen, K.G., Legay, A., Nyman, U., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transfer **17**(4), 397–415 (2015)
9. Eckersley, P.: How unique is your web browser? In: International Symposium on Privacy Enhancing Technologies Symposium, pp. 1–18. Springer (2010)
10. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University, USA (2004), aAI3114521
11. Fang, Y., Huang, C., Zeng, M., Zhao, Z., Huang, C.: Jstrong: malicious javascript detection based on code semantic representation and graph neural network. Comput. Secur. **118**, 102715 (2022)
12. Federal trade commission: as nationwide fraud losses top $10 billion in 2023, FTC steps up efforts to protect the public. https://www.ftc.gov/news-events/news/press-releases/2024/02/nationwide-fraud-losses-top-10-billion-2023-ftc-steps-efforts-protect-public, February 2024, Accessed 29 Sep 2024
13. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 1–12. ACM (2006)
14. Fouad, I., Santos, C., Legout, A., Bielova, N.: Did I delete my cookies? cookies respawning with browser fingerprinting. CoRR **abs/2105.04381** (2021), https://arxiv.org/abs/2105.04381
15. Gk: browser fingerprinting: an introduction and the challenges ahead: Tor project, September 2019, https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead/
16. Hiremath, P.N., Armentrout, J., Vu, S., Nguyen, T.N., Minh, Q.T., Phung, P.H.: Mywebguard: toward a user-oriented tool for security and privacy protection on the web. In: Dang, T.K., Küng, J., Takizawa, M., Bui, S.H. (eds.) Future Data and Security Engineering, pp. 506–525. Springer, Cham (2019)
17. Iqbal, U., Englehardt, S., Shafiq, Z.: Fingerprinting the fingerprinters: learning to detect browser fingerprinting behaviors. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 1143–1161. IEEE (2021)

18. Iqbal, U., Snyder, P., Zhu, S., Livshits, B., Qian, Z., Shafiq, Z.: Adgraph: a graph-based approach to ad and tracker blocking. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 763–776. IEEE (2020)
19. Karami, S., Ilia, P., Solomos, K., Polakis, J.: Carnus: exploring the privacy threats of browser extension fingerprinting. In: Proceedings of the 27th Network and Distributed System Security Symposium (NDSS) (2020)
20. Khademi, A.F., Zulkernine, M., Weldemariam, K.: An empirical evaluation of web-based fingerprinting. IEEE Softw. **32**(4), 46–52 (2015)
21. Lanze, F., Panchenko, A., Engel, T.: A formalization of fingerprinting techniques. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 818–825. IEEE (2015)
22. Laperdrix, P.: Morellian analysis for browsers (2019). https://github.com/plaperdr/morellian-canvas
23. Laperdrix, P.: Learn how identifiable you are on the internet (2024). https://amiunique.org/
24. Laperdrix, P., Avoine, G., Baudry, B., Nikiforakis, N.: Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 43–66. Springer (2019)
25. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: a survey. ACM Trans. Web **14**(2) (2020). https://doi.org/10.1145/3386040
26. Laperdrix, P., Rudametkin, W., Baudry, B.: Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 98–108. IEEE (2015)
27. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. In: International Journal on Software Tools for Technology Transfer, vol. 1, pp. 134–152. Springer (1997)
28. Li, S., Cao, Y.: Who touched my browser fingerprint? a large-scale measurement study and classification of fingerprint dynamics. In: Proceedings of the ACM Internet Measurement Conference, pp. 370–385 (2020)
29. Mowery, K., Shacham, H.: Pixel perfect: fingerprinting canvas in html5. In: Proceedings of W2SP, vol. 2012 (2012)
30. Nguyen, L.V., Mohan, G., Weimer, J., Sokolsky, O., Lee, I., Alur, R.: Detecting security leaks in hybrid systems with information flow analysis. In: Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, pp. 1–11 (2019)
31. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, pp. 47–60 (2009)
32. Pugliese, G., Riess, C., Gassmann, F., Benenson, Z.: Long-term observation on browser fingerprinting: users' trackability and perspective. Proc. Priv. Enhancing Technol. **2020**(2), 558–577 (2020)
33. Queiroz, J.S., Feitosa, E.L.: A web browser fingerprinting method based on the web audio API. Comput. J. **62**(8), 1106–1120 (2019)
34. Reynolds, M.: An axiomatization of full computation tree logic. J. Symbol. Logic **66**(3), 1011–1057 (2001)
35. Senol, A., Ukani, A., Cutler, D., Bilogrevic, I.: The double edged sword: identifying authentication pages and their fingerprinting behavior. In: The Web Conference (WWW), vol. 2024 (2024)
36. Team, B.P.: Fingerprint randomization, June 2020. https://brave.com/privacy-updates/3-fingerprint-randomization/

37. Team, F.: Fingerprintjs. https://github.com/fingerprintjs/fingerprintjs (2024). Accessed 29 Sep 2024
38. UNION, E.: Directive 2009/136/ec of the European parliament and of the council. Official J. Eur. Union **337**, 11 (2009)
39. Vastel, A., Laperdrix, P., Rudametkin, W., Rouvoy, R.: Fp-stalker: tracking browser fingerprint evolutions. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 728–741. IEEE (2018)
40. Zhang, D., Zhang, J., Bu, Y., Chen, B., Sun, C., Wang, T., et al.: A survey of browser fingerprint research and application. Wirel. Commun. Mob. Comput. **2022** (2022)