Deterministic Browser

I. Introduction

[1]

II. Deterministic vs. Stochastic (Probabilistic)

Traditionally, there are two categories of approaches in defending browser fingerprinting: normalizing every fingerprint-able output (we call it deterministic output), and randomizing the output (we call it stochastic output). However, neither approach is the key in defending fingerprinting. Let us find the key. Consider a fingerprint-able output as a random variable with the following distribution function:

$$f(o, x_1, x_2, ..., e) = Pr(O, X_1, X_2, ..., E) \quad (1)$$

where o is the fingerprint-able output, x_1, x_2,..., are different parameters for the output, and e is the runtime environment to generate the output. For example, $f(Interval, op, E)$ is the probability of an opcode op taking $Interval$ to execute in E; $f(Interval, IP, E)$ is the probability of a network request to IP taking $Interval$ to finish in E; $f(Image, Inputs, E)$ is the probability of a canvas rendering an $Image$ with $Inputs$ in E.

The reason of a fingerprint-able output o is that $f(o, x_1, x_2, ..., E^1)$ and $f(o, x_1, x_2, ..., E^2)$ differ for E^1 and E^2. Based on the differences, one can recognize the corresponding E. For a simplified example, in E^1, an opcode op has 0.5 probability to finish in 0.1ms, and 0.5 probability to finish in 0.2ms; at contrast, in E^2, an opcode op has 0.5 probability to finish in 0.2ms, and 0.5 probability to finish in 0.3ms. Based on average finishing time (0.15ms vs. 0.25ms), one can differentiate E^1 and E^2.

Therefore, we believe that the key of defending fingerprinting lies in that the distribution functions for a fingerprint-able output are the same over different Es. That is,

$$\forall E^1, \forall E^2, f(o, x_1, x_2, ..., E^1) = f(o, x_1, x_2, ..., E^2) \quad (2)$$

Now, let us look back at deterministic and stochastic approaches. Stochastic approach itself does not defend fingerprinting: Only when the distribution function for a stochastic output is deterministic, such approach can defend fingerprinting. This is also the common mistake that existing approaches make in defending fingerprinting. For example, Tor Browser [] adds jitters to $Date$ to defend JavaScript performance fingerprinting, however the browser can still be fingerprinted in a long run when the jitter is averaged out. Similarly, PriVaricator [] adds randomized noise to certain fingerprint-able outputs, however the browser can still be fingerprinted if one obtains the outputs many times and calculates the average.

Deterministic approach can defend fingerprinting, because a deterministic output is a special case of stochastic approach with determined distribution function (at one point, f equals 1, and at other points, f equals 0).

Next, the question is whether to use deterministic or stochastic approach in defending fingerprinting. Because stochastic approach does not directly defend fingerprinting, it is unnecessary to randomize environment-specific constant (such as browser version) and introduce deterministic distribution function, which adds overhead and causes compatibility issues. Therefore, we mainly adopt determinism to combat fingerprinting.

Theoretically, we could make a fully deterministic browser mitigating all the stochastic factors. However, for similar reasons, it is unnecessary to fix the value for a random variable especially when the variable has a deterministic distribution function. A simple example is that there is no need to get rid of the random number generator in the JavaScript $Math$ object. That is, precisely, our approach is a *mostly deterministic browser where stochastic factors have a deterministic distribution function*, for short, deterministic browser.

III. Definition of Determinism

In this section, we are going to formally define the terminology—determinism—in the context of a program. Determinism for a single operation is as simple as normalizing the output for the operation. Such determinism has already been implemented by Tor Browser [], e.g., fetching a normalized user agent. However, it is unclear how to define determinism for an entire program. Below, we will give two definitions of determinism: one is strong and one weak.

Let us first present some background information. Before a browser executes a program, such as JavaScript, the browser—particularly the JavaScript engine part—will parse the JavaScript program and convert the program to a special form called operation code (opcode). Such opcodes are specific to a certain browser, such as IE, Firefox, and Google Chrome, and such conversion from a JavaScript program to opcodes is unique once the browser is fixed. In our definition of determinism, we can choose any set of opcodes. Now let us define determinism below:

Definition 1 *Given a set of opcodes (SO) generated from a program, a fixed initial state, and a set of fixed inputs (I) happening at fixed timestamps,* **strong determinism** *defines that for different executions on different environments, at time t, the same opcode (op) will be executed and the same outputs ($O_{op,I}$) will be produced.*

Strong determinism states that when inputs for a browser are fixed, every JavaScript operation will produce the same results at the same timestamp in different environments. That is, when

a snippet of JavaScript tries to fingerprint the environment that it is executing upon, the JavaScript cannot differentiate the environments as they look exactly the same from the viewpoint of the JavaScript.

Strong determinism can defend browser fingerprinting, however it assumes that every execution starts at the same timestamp—being unrealistic for modern browsers because in real world the time elapses and never comes back. Therefore, we propose weak determinism, a variance of strong determinism but taking different starting timestamps into consideration.

Definition 2 *Given a set of opcodes (SO) generated from a program, a fixed initial state, and a set of fixed inputs (I),* **weak determinism** *defines that for any two executions (E^1 and E^2) on different environments, if each $i \in I$ happens at t_i^1 and t_i^2 in E^1 and E^2, and $t_i^1 - t_i^2 = C$, where C is a constant related to only E^1 and E^2 but not i, the followings hold:*

(1) for every opcode (op) in E^1 and E^2, $t_{op}^1 - t_{op}^2 = C$, where t_{op}^k is the timestamp when the opcode op is executed in E^k;

(2) for every opcode (op) in E^1 and E^2, $O_{op,I}^1 - O_{op,I}^2 = f(t_{op}^1) - f(t_{op}^2)$, where f is a function only related to SO. Note that in most cases, $O_{op,I}^1 = O_{op,I}^2$.

Weak determinism states that when inputs are fixed and span over time in a fixed pattern, JavaScript operations in different executions will follow a determined distribution pattern over time axis and produce almost the same results. Now, let us take a closer look at these two properties. First, although the starting timestamps in different executions are different, the execution patterns of a specific JavaScript program are the same. That is, if one makes a translation from one execution to the starting point of another, the two executions look the same. Second, the outputs of opcodes in different executions are the same except when the opcodes are related to the absolute time of the execution.

Weak determinism is weaker than strong determinism because a JavaScript program can still differentiate executions based on the starting time. However, the JavaScript program cannot differentiate the runtime environment, i.e., the program could be executed on the same browser at different timestamps, or on different browsers at different timestamps. As the starting timestamp is important to a JavaScript program but not used in fingerprinting browsers, in the rest, we refer determinism as the weak determinism defined in this section.

IV. Reference Frames and Clocks

Determinism defines fixed execution patterns for a program in a browser. One intuitive method is to arrange the opcode execution sequence following a pre-determined pattern. However, such arrangement will slow down a faster execution, and cannot make up for a slower execution (as the clock does not wait for a slower execution). Therefore, instead of changing the execution, we change the clock of a browser so that the perceived execution pattern from the viewpoint of the JavaScript program is fixed, although we, as oracles, will see the execution pattern differently. This leads to a new concept called *reference frames*.

The concept of reference frame, borrowed from Physics, consists of "an abstract coordinate system and the set of physical reference points that uniquely fix the coordinate system and standardize measurements." In the context of deterministic browser, a reference frame is one-dimension system and consists of only time. The measurement of time is performed by *clocks*, which ticks based on certain criteria, such as the physical clock, the execution of an opcode, and the invocation of a document object model (DOM) method.

We further classify reference frames into two major categories: main and auxiliary. A main reference frame has an *observer*, a Turing complete program, e.g., capable of fingerprinting the runtime environment; an auxiliary reference frame does not have an observer. Correspondingly, we define a clock in a main reference frame as main clock, and a clock in auxiliary as auxiliary clock.

A main reference frame can exist by itself; on the contrary, an auxiliary reference frame does not exist by itself, but is attached to a main reference frame. That is, an auxiliary reference frame is created and destroyed by a main reference frame by operations. For example, when a JavaScript sends a network request, a networking auxiliary reference is created; when the JavaScript receives the response, the networking auxiliary reference is created. Lacking of observers, an auxiliary reference frame cannot fingerprint runtime environment, however it can convey information such as auxiliary clocks to the main reference frame.

Now, let us study the relationship between reference frames and determinism. We say that a main reference frame is deterministic, if the observer obeys the properties of determinism defined in Section III under the clock of the reference frame. That is, from the observer's viewpoint, the execution of opcodes distributed over time is in a fixed pattern, and the outputs of each opcode are only related to the starting time, initial state, and inputs.

Because there are no observers in auxiliary reference frames, the determinism of an auxiliary reference frame is related to its parent main reference frame. That is, if the operations of the observer in the parent reference frame—which interact with the auxiliary reference frame, such as creation and destroying—obeys the determinism properties under the clock of the auxiliary reference frame, the auxiliary reference frame is deterministic.

Below, we will take a look at some reference frame examples, how the clock ticks in these reference frames, and whether these reference frames are determinstic.

Example 1: JavaScript Main Reference Frame. JavaScript program resides in a main reference frame, because JavaScript is a Turing complete language capable of fingerprinting the runtime environment. The main clock ticks based on the following two criteria:

(i) When there are opcodes running, the clock ticks with regards to the executed opcode. That is, we have the following

equation,

$$t_{now} = t_{start} + \sum_{op \in EJO} unit_{op} \tag{3}$$

where EJO is the set of executed JavaScript opcodes and $unit_{op}$ is the atomic elapsed time for that opcode. If we normalize $unit_{op}$, Equation 3 becomes Equation 4:

$$t_{now} = t_{start} + count_{EJO} \times unit \tag{4}$$

(ii) When there are no opcodes running, the clock ticks with regards to the physical clock:

$$t_{now} = t_{start} + \Delta t_{physical} \tag{5}$$

These two criteria are applied together in the reference frame. Say, a JavaScript program are first executed, and the main clock ticks based on the amount of executed opcodes. Then, the JavaScript program stops for events, such as setTimeout, during which the main clock ticks based on the physical clock.

Now, let us see why the defined clock makes the reference frame deterministic. We will look at the two properties of determinism separately. First, assume that we have two executions of the same JavaScript at t^1_{start} and t^2_{start} separately. For an arbitrary opcode op, we will have Equation 6.

$$\begin{aligned} & t^1_{op} - t^2_{op} \\ & = (t^1_{start} + \Delta t_{physical,idle} + count_{EJO} \times unit) \\ & - (t^2_{start} + \Delta t_{physical,idle} + count_{EJO} \times unit) \\ & = t^1_{start} - t^2_{start} = C \end{aligned} \tag{6}$$

In Equation 6, EJOs for two different executions are the same due to the inherent property of JavaScript engine: The same JavaScript engine of a browser always generates the same sequence of opcodes for a given JavaScript program. Further, the idle time intervals are also the same when the inputs and distribution patterns are fixed. Therefore, the reference frame satisfies the first property.

Then, we will see whether the reference frame satisfies the second property, which says that given fixed inputs, every opcode produces the same results. If the opcode does not involve external objects, the results remain the same for different executions. For example, one plus one always equals two in different executions. If the opcode involves external objects, the results may change in different environments, e.g., $canvas$ object may render contents differently with different operating systems. In Section VII, we will talk about how to normalize outputs using virtualization. For now, we first assume that the reference frame has this property, and will leave the details in Section VII.

Example 2: DOM Auxiliary Reference Frame. A DOM auxiliary reference frame is attached to a JavaScript main reference frame: Such reference frame is created by the invocation of DOM operation via JavaScript and destroyed when the DOM operation finishes. For example, when a JavaScript creates an image tag, a DOM auxiliary reference frame is created; when the image tag is loaded and the corresponding $onload$ event is triggered, the DOM auxiliary reference frame is destroyed.

The clock in a DOM auxiliary reference frame ticks based on atomic DOM operations, such as the creation and deletion of a DOM element. DOM auxiliary reference frames are deterministic, because given inputs specific, such as a web page, DOM operations, such as loading the page, will finish in a determined interval.

Example 3: Networking Auxiliary Reference Frame. A networking auxiliary reference frame is attached to the JavaScript main reference frame as well. Networking auxiliary reference frames are created by an HTTP request, and destroyed by an HTTP response, both initiated from the JavaScript main reference frame.

The clock in a networking auxiliary reference frame ticks based on physical clock, and is not deterministic. Instead, as discussed in Section II, we adopt a random variable with a deterministic distribution function instead of determinism. Specifically, we use Tor network that randomizes the network delay between the request and the response.

The reason of such clock is that network delay is undetermined: If we make it deterministic (though theoretically possible), as discussed later in Section V, we need to pause the main reference frame clock and delay the JavaScript execution. Instead, we rely on an existing approach, Tor Network, to achieve anonymization in the network layer, which can be classified a deterministic distribution function approach.

Example 4: Keyboard Auxiliary Reference Frame. A keyboard auxiliary reference frame, attached to the JavaScript main reference frame, is created when a keyboard event is registered, and destroyed together with the main reference frame, i.e., sharing the same duration of the Web frame. The clock in a keyboard auxiliary frame ticks based on two criteria: If keystrokes come consecutively in an interval less than a threshold, the clock ticks based on the number of keystrokes; otherwise, the clock ticks based on physical clock. Such reference frame is deterministic in small scale (when a person keeps typing), and stochastic with a deterministic distribution function in large scale. We choose such clock because the typing behavior of a person in small scale is fingerprint-able (i.e., with different typing speed).

V. Reference Frame Communication

In previous section, we have discussed reference frames and how clocks tick in different reference frames. Because the measurements of time are different in these reference frames, we do not want to reveal the clock of one frame to another, especially when one of the clocks ticks based on the physical clock. Now, we will introduce how to synchronize these clocks when reference frames communicate with each other.

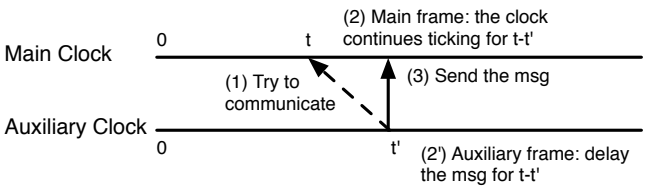

Fig. 1: Clock Synchronization between Main and Auxiliary Reference Frames.

A. Communication between Main and Auxiliary Reference Frames

Main and auxiliary reference frames talk to each other during circumstances such as creation and destroying of an auxiliary reference frame. As mentioned, because there are no observers in auxiliary reference frames, we focus on the case when an auxiliary reference frame conveys the clock to the main reference frame, such as a response message from a networking auxiliary reference frame to the JavaScript in the main reference frame, and a keystroke from the keyboard auxiliary reference frame to the JavaScript.

Figure 1 shows the communication between two reference frames. First, the auxiliary reference frame tries to send a message to the main reference frame at time t' of the auxiliary clock. However, in the main clock, the time is only t, slower than the auxiliary clock. Therefore, instead of sending the message, we will delay the message for an interval of $t' - t$, and then deliver the message at time t of the main clock. That is, in the viewpoint of the message, these two clocks are synchronized: The main reference frame sends the message at time t', and the auxiliary reference frame receives the message at time t' as well.

There is a prerequisite in the aforementioned communication: The main clock is slower than the auxiliary clock. This prerequisite is easy to achieve as we

Let us take a look at several examples.

Why not determined

Clock fast forwarding

B. Synchronization between Two Main Reference Frames

VI. Randomization

VII. Virtualization: JavaScript Object

VIII. How to defend

JavaScript performance

Network difference

Keystroke

Canvas

References

[1] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *NDSS*, 2015.