Securing Timeout Instructions in Web Applications

Alejandro Russo   Andrei Sabelfeld
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract

Timeout mechanisms are a useful feature for web applications. However, these mechanisms need to be used with care because, if used as-is, they are vulnerable to timing attacks. This paper focuses on internal timing attacks, *a particularly dangerous class of timing attacks, where the attacker needs no access to a clock. In the context of client-side web application security, we present JavaScript-based exploits against the timeout mechanism of the DOM (document object model), supported by the modern browsers. Our experimental findings reveal rather liberal choices for the timeout semantics by different browsers and motivate the need for a general security solution. We propose a foundation for such a solution in the form of a runtime monitor. We illustrate for a simple language that, while being more permissive than a typical static analysis, the monitor enforces termination-insensitive noninterference.*

1. Introduction

Timeout mechanisms provide a useful feature for web applications. For example, they are helpful for many client-side scenarios, ranging from providing handlers when there is a network problem to controlling the timing of slide shows, automatically closing a window after a given period of time, etc.

However, these mechanisms need to be used with care because, if used as-is, they are vulnerable to timing attacks. These attacks exploit the information flow from secrets to the execution time. *Internal timing attacks* [54] go one step further and exploit the flow from execution time to publicly observable events. These attacks are particularly dangerous because the attacker needs no access to a clock to learn the complete secrets in linear time [36].

While external timing leaks have been explored in the context of web applications (e.g., [17]), we turn our attention to internal timing leaks for web clients. We are particularly interested in protecting the timeout model that is available via the API of the DOM (document object model), as supported by modern browsers.

To demonstrate the problem, we present JavaScript-based exploits against the timeout mechanism of the DOM for a collection of modern browsers. The exploits also succeed against an information-flow enhanced extension of Firefox by Vogt et al. [52], whose information-flow tracking is not sufficient against timing attacks.

Our experimental findings reveal rather liberal choices for the timeout semantics by the different browsers, which motivates the need for a general security solution. We propose a foundation for such a solution in the form of a runtime monitor. The intention for this kind of monitor is to be deployed in a browser to track manipulation of sensitive data such as form input, cookies, browsing history, etc. to make sure this data is not leaked to the attacker (for example, by encoding it into the URL of an embedded image from a third-party web site).

The paper works out the idea of tracking information flow in the presence of timeouts for a simple imperative language. The monitor keeps track of security levels (in a simple setting, *secret* and *public*) for variables. The monitor prevents insecure flow of information via assignments to public variables, making sure (i) expressions on the right hand side of assignments do not depend on secrets (*explicit flows*) and (ii) these assignments are not made in a *secret context*, i.e., inside of a conditional or a loop with a secret guard. The latter corresponds to *implicit flows* [16].

In addition to checking these flows, the monitor keeps track of the commands in the timeout queue. These commands are not allowed to assign to public variables if their respective timeouts were set in a secret context or the time depended on secrets. In addition, whenever the main computation branches on secret data, the commands in the queue are “tainted”: their future attempts of performing a publicly observable assignment will be stopped. Also, when a tainted timeout executes, timeouts to be created in the future are also tainted. We show that this simple mechanism, together with preventing timeout setting by commands that were tainted while in the timeout queue, is sufficient to guarantee information-flow security.

We show that, while being more permissive than a typical static analysis, the monitor prevents insecure information flow. More precisely, we show that the monitor enforces termination-insensitive noninterference [5]. The permissiveness of the monitor is particularly important in the context of web applications, where freshly generated code is often dynamically evaluated. Clearly, it is quite a challenge to avoid a significant loss of precision when analyzing such code statically.

The paper is organized as follows. In Section 2, we reflect on static vs. dynamic information-flow enforcement. We present JavaScript-based attacks in Section 3. Section 4 formalizes the semantics of a simple language with timeouts. Section 5 presents the monitor. Section 7 proves the security of the monitor. Section 6 evaluates how a full version of our monitor for JavaScript would behave for some common uses of timeouts. Section 8 discusses related work. The paper concludes with Section 9.

2. Static vs. dynamic enforcement

Before getting into the specifics of the timeout issues, we would like to offer some reflections on a general discussion on static vs. dynamic enforcement for information-flow control. A separate paper elaborates on these reflections and formalizes them for a simple imperative language [42].

Historically, dynamic techniques are the pioneers of the area of information flow in the 70's (e.g., [18]). They prevent explicit flows (as in $public := secret$) in program runs. They also address implicit flows (as in $\texttt{if}\ secret\ \texttt{then}\ public := 1$) by enforcing a simple invariant of no public side effects in *secret context*, i.e., in the branches of conditionals and loops with secret guards. These techniques, however, come without soundness arguments.

In their seminal paper, Denning and Denning [16] suggest a static alternative for information-flow analysis. They argue that static analysis removes runtime overhead for security checks. This analysis prevents both explicit and implicit flows statically. The invariant of no public side effects in secret context is ensured by a syntactic check: no assignments to public variables are allowed in secret context. Denning and Denning do not discuss soundness, but Volpano et al. [55] show soundness by proving termination-insensitive noninterference, when they cast Denning and Denning's analysis as a security type system. Denning-style analysis is by now the core for information-flow tools Jif [28], FlowCaml [45], and the SPARK Examiner [9], [14].

The 90's see the domination of static techniques for information flow [41]. The common wisdom appears to be that dynamic approaches are not a good match for security since monitoring a single path misses public side effects that could have happened in other paths.

For example, Myers and Liskov [27] discuss:

> . . . static checking allows precise, fine-grained analysis of information flows, and can capture implicit flows properly, whereas *dynamic label checks* create information channels that *must be controlled through additional static checking*. . .

It is, in effect, suggested that dynamic checking alone is insufficient for security.

In this light, it might be surprising that it is possible for purely dynamic enforcement to be *as secure as Denning-style static analysis* [42]. The key factor is termination. Denning-style static analysis are typically termination-insensitive (i.e., they ignore leaks via the termination behavior of the program). Thus, they satisfy termination-insensitive noninterference [55], which ignores the channel for signals via the (non)termination of the program. If the monitor, by stopping the underlying program, can introduce nontermination, this feature can be used for collapsing information channels into the termination channel. The implicit-flow channel is one example: stopping the execution at an attempt of a public assignment in secret context (note the similarities to the techniques from the 70's!) is in fact sufficient for termination-sensitive security.

Static techniques have benefits of reducing runtime overhead, and dynamic techniques have the benefits of permissiveness, which is of particular importance in dynamic applications, where freshly generated code is evaluated. For example, it is difficult to statically determine if the program $\texttt{eval}(\texttt{http://www.dynamic.com/script.js})$ is secure without being too conservative (command `eval` is used here to retrieve and execute a script under a given URL). The difficulty arises from the fact that the code retrieved and executed from $\texttt{http://www.dynamic.com/script.js}$ might change over time. Moreover, in a heterogeneous environment as the web, it is also difficult to assume properties about third-parties scripts. Another example to illustrate permissiveness of dynamic techniques is the program $\texttt{if}\ l < 0\ \texttt{then}\ l := 1\ \texttt{else}\ l := h$, where

l and h are variables that store public and secret values, respectively. Static analysis, as traditional type systems [55], reject this program as insecure due to the presence of the explicit flow $l := h$. In contrast, some dynamic techniques, as the monitor introduced in Section 5, are able to accept executions of the program when $l < 0$ holds. On the security side, however, both Denning-style analysis and dynamic enforcement have the same guarantees: termination-insensitive noninterference.

When termination-sensitive noninterference is desired, the absence of side effects of traces not taken becomes indeed hard to guarantee dynamically.

But which policy should be the one of choice, termination-insensitive noninterference or termination-sensitive noninterference? Termination-sensitive noninterference is attractive, but rather difficult to guarantee. Typically, strong restrictions (such as no loops with secret guards [53]) are enforced. Program errors exacerbate the problem. Even in languages like Agda [30], where it is impossible to write nonterminating programs, it is possible to write programs that terminate abnormally: for example, with stack overflow. Generally, abnormal termination due to resource exhaustion, is a channel for leaks that can be hard to counter.

The above-mentioned information-flow tools Jif [28], FlowCaml [45], and the SPARK Examiner [9], [14] avoid these problems by targeting termination-insensitive noninterference. The price is that the attacker may leak secrets by brute-force attacks via the termination channel. But there is formal assurance that that these are the only possible attacks. Askarov et al. [5] show that if a program satisfies termination-insensitive noninterference, then the attacker may not learn the secret in polynomial running time in the size of the secret; and, for uniformly-distributed secrets, the probability of guessing the secret in polynomial running time is negligible.

Having outlined the space of choices, we chose termination-insensitive noninterference as the target security policy, similarly to Jif, FlowCaml, and SPARK Examiner. Further, we chose dynamic enforcement for our enforcement method, driven by the need for permissiveness (especially important for web applications).

3. Attacks

This presents the attacker model and providing examples of client-side JavaScript-based attacks that involve internal timing leaks.

Attacker model The attacker's target is user-sensitive data that is available to the browser in the context of a given web page or data stored at the server that might be accessible in the context of user sessions. This data includes browser cookies, form input, browsing history, etc. (cf. the list of sensitive sources used by Netscape Navigator 3 [29]). Client-side scripts have full access to such data. In that way, scripts can perform useful computations on the client-side before sending requests to the server. An example is data validation, where scripts check, for example, that bank account numbers have the proper length and clearing numbers. It is important to guarantee that scripts preserve confidentiality properties, i.e., they do not leak information by transferring secret data into public sinks. We assume that public sinks are observable by the attacker, which includes communications to attacker-observable web sites. Note that requesting information from the attacker's web site is sufficient to transfer information as the information may be encoded in the URL of the request. We adopt the worst-case assumption that the attacker has full control over client-side code independently of the origin. This captures a wide range of attackers, including those that succeed in taking over the control of the client-side code by cross-site scripting (XSS).

Timeouts According to [2], timeouts execute code snippets, or functions, after a specified delay. This feature is commonly used to address possible failing XMLHTTP requests in web applications. However, it is also used for other purposes: animations, web slide shows, etc. Timeouts are provided by the DOM API. Although timeouts are present in most web browsers, this feature is not yet part of any standard. As of now, it has only been considered in drafts for the HTML 5 standard. We found different implementations of the API in different browsers. Some of these discrepancies can indeed be exploited to leak secrets, as discussed below. For the rest of the section, we assume that JavaScript is the language to interact with the DOM API.

Timeouts can be used to leak sensitive data. Before illustrating it, we describe in more detail how timeouts are handled by modern web browsers. As a piece of JavaScript runs, it may set timeouts. The first timeout is triggered after finishing the execution of the JavaScript present in the web page. The next timeout, if there is more than one, is triggered after the first timeout completes its execution. In other words, JavaScripts included in web pages cannot be interrupted by timeouts as well as timeouts cannot be interrupted by other timeouts. With this in mind, in Figure 3, we present a JavaScript attack based on timeouts. We consider

```
function l0() { l = 0 ; }                  function l1() { l = 1 ; }

                                           function f() {
function attack() {                          var z = 0 ;
  setTimeout(function()l0(),50);             if (form.secret > 0)
  setTimeout(function()f(),1);                 {do {z++;} while( z < 900000);}
  setTimeout(function()leak(),500); }        else {};
                                             setTimeout(function(){l1();},1); }

 function leak { new Image().src="http://www.evil.com/leak="+encodeURI(l); }
 attack();
```

Fig. 1. Internal timing leak

one-bit secret represented by `form.secret`, a local secret variable `z`, and a public variable `l`. Observe that `l` is only assigned to constant values. One of the peculiarities of this attack is the absence of *explicit* and *implicit* flows [16]. Nevertheless, it does produce a leak by transferring secret information into `l`. The attack starts by running function `attack` and setting up three timeouts: a timeout for calling `l0()`, which sets `l` to zero, a timeouts that inspect the value of the secret (function `f`), and a timeout that sends `l` to the attacker's site (function `leak`), which we assume is placed in a different origin from the one where the script runs. After that, the function associated with the minimum delay is run, in this case, function `f`. Remember that we still have the timeout that runs `l0()` scheduled to run after 50 millisecond since it was set. Then, function `f` inspects the value of `form.secret`. If `form.secret` > 0, then it performs some dummy computations that take around 100 milliseconds (see the loop in the code) [1]. Otherwise, no computations are performed at all. After that, `f` sets a timeout to execute function `l1()` after 1 millisecond. On one hand, if `form.secret` > 0, function `l1()` is executed after `l0()` since 50 milliseconds already passed since the timeout for `l0()` was set. On the other hand, if `form.secret` ≤ 0, `l1()` is executed first because less than 50 millisecond passed since setting the timeout for `l0()`. Observe that depending on the value of `form.secret`, which affects the timing behavior of function `f`, the race to assign a value to `l` is resolved in different manners. More fundamentally, our attack produces an internal timing leak: a leak that is created by exploiting timing behavior of programs without needing access to a stopwatch. It is possible to magnify the attack in Figure 3 to leak the complete value of secrets [36]. Observe that it is usually enough two conditions to perform this attack on a given web site. Firstly, JavaScript, and particularly timeouts, must be allowed to run. Secondly, the code in Figure 3 should be somehow present in the website. It could be the case that the code was placed by a malicious programmer or injected by an attacker [4]. To the best of our knowledge, this work is the first one to address this kind of attacks in web applications.

In Figure 3, we present another example where timeouts exploit a different covert channel: termination. Variable `nonexist` is a non-declared variable. As a consequence of that, when running the line `nonexist` $= 0$, the execution of the JavaScript will be aborted. However, timeouts set to run will be triggered anyway. The program essentially terminates successfully when `form.secret` ≤ 0 and abnormally otherwise. Then, by inspecting the value of `l`, it is possible to distinguish between these two cases. As for the example in Figure 3, it is possible to magnify this termination attack to leak complete secret values [36].

Implementation in web browsers As mentioned previously, timeouts are not yet part of any standard. However, most browsers support them. In this light, we experiment with several browsers to try to find similarities and differences between them. In Figure 3, we present a JavaScript program with two meta-variables d_1 and d_2 to represent delays for the given timeouts. We assume that the total time to run function `foo` is 200 milliseconds. We also assume that $d_1 > d_2$. It turns out that how much d_1 and d_2 are different does not matter. Figure 4 presents a table that reveals, depending on the values of d_1, d_2, and the web browser, different behaviors when running `foo`(). For instance, the first column indicates that, in Firefox, when d_1 and d_2 are strictly less than 11 milliseconds, function `f` runs first. The second column indicates that when d_1 and d_2 are bigger than the total time that takes to run `foo()`, function `g` runs first in most of the browsers except for Konqueror. The choice of Konqueror is somewhat anomalous because

1. The scripts have been run in the following computers: AMD Athlon 64 3200+, 1GB RAM with the Gentoo Linux distribution, and Intel P4 2,6GHz, 1GB RAM with Microsoft Windows XP.

```
function attack() {
  setTimeout(function()leak(),1);
  if (form.secret > 0) nonexist = 0 ;
  l = 0 ;}
function leak { new Image().src=
"http://www.evil.com/leak="+encodeURI(l); }
l = 1; attack();
```

Fig. 2. Termination leak

```
function foo() {
setTimeout(function()f(),d1);
setTimeout(function()g(),d2);
// Some code
}
foo() ;
```

Fig. 3. Controversial timeout example

Browser	d_1, d_2	$d_i > 200ms$
Chrome 1.0.154.43	$<10ms$, f	g
Firefox 3.0.3	$<11ms$, f	g
Konqueror 3.5.9	$<5ms$, f	f
Explorer 7.0.5730.11	g	g
Opera 9.62	g	g

Fig. 4. Which function runs first?

```
function attack() {
  var z = 0;
  setTimeout(function()l0(),80);
  setTimeout(function()l1(),50);
  if (form.secret > 0) ;
    {do {z++;} while( z < 900000);}
  else {};
}
setTimeout(function()leak(),500) ;
attack();
```

Fig. 5. Attack for Konqueror

$$e ::= n \mid x \mid e \oplus e$$

$$c ::= \texttt{skip} \mid x := e \mid c;c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ b\ \texttt{do}\ c \mid \mathit{end} \mid \texttt{setTimeout}(c,e) \mid \mathit{stop}$$

Fig. 6. Language

expired timeouts are ordered not according to the time parameters but according to *order of creation*. One could argue that we might not bother about this anomalous semantics, but we nevertheless discuss how to model and secure this choice by adapting our approach. Konqueror's apparently innocent choice leads to the possibility to create another attack that uses the internal timing covert channel. We show such an attack in Figure 5. Functions `l0`, `l1`, and `leak` are the same as the ones described in Figure 3. When `form.secret` > 0, function `attack` takes at least 100 millisecond due to the while-loop in the code, which produces function `l0` to run first. Otherwise, function `l1` executes first due to the running time of `attack` is less that 50 milliseconds. In the code, the interesting feature of this attack is the absence of public observable events after branching on secrets. Observe that the branching in function `f` is followed by no instructions. Some type systems [12], [13] prevent internal timing leaks by restricting instructions after branching on secrets. Unfortunately, these type systems miss this attack because there are no instructions after branchings. However, our proposed solution in Section 5 can be easily adapted to prevent it.

4. Semantics

Language Figure 6 presents a simple imperative language with a timeout primitive. Expressions e consist of integers n, variables x, and composite expressions $e \oplus e$ (where $\oplus$ is a binary operation). Commands, denoted by c, consist of standard imperative instructions and `setTimeout`. Instruction $\texttt{setTimeout}(c, e)$ establishes a code snippet that will run after e units of time. The language contains additional commands signifying an end of a structure block (end) and termination (stop), explained below. These additional commands can be generated during the execution but they are not used in initial configurations (this restriction is formalized in Section 7). A command c, memory m, current time t, and a list of timeouts p form a *command configuration* $\langle c, m, t, p\rangle$. Lists of timeouts are composed by elements of the form (c, t) where c is a snippet to be run at time t. We only consider lists of timeouts in ascending order of time. Small-step semantics is described by transitions of the form $\langle c, m, t, p\rangle \xrightarrow{\alpha}_{\gamma} \langle c', m', t', p'\rangle$, where α is an *internal* event triggered by the transition. The internal event label serves the purpose of conveying information about program execution to an execution monitor. As we explain in Section 5, the monitor uses this information in order to determine if the execution can proceed. On the other hand, γ is an *external* event.

$$\frac{t' = t + T(\texttt{skip})}{\langle \texttt{skip}, m, t, p\rangle \xrightarrow{s} \langle \mathit{stop}, m, t', p\rangle}$$

$$\frac{m(e) = n \qquad t' = t + T(x := e)}{\langle x := e, m, t, p\rangle \xrightarrow{a(x,e)}_{(x,n)} \langle \mathit{stop}, m[x \mapsto n], t', p\rangle}$$

$$\frac{\langle c_1, m, t, p\rangle \xrightarrow{\alpha}_{\gamma} \langle \mathit{stop}, m', t', p'\rangle}{\langle c_1; c_2, m, t, p\rangle \xrightarrow{\alpha}_{\gamma} \langle c_2, m', t', p'\rangle}$$

$$\frac{\langle c_1, m, t, p\rangle \xrightarrow{\alpha}_{\gamma} \langle c_1', m', t', p'\rangle \qquad c_1, c_1' \neq \mathit{stop}}{\langle c_1; c_2, m, t, p\rangle \xrightarrow{\alpha}_{\gamma} \langle c_1'; c_2, m', t', p'\rangle}$$

$$\frac{m(e) \neq 0 \qquad t' = t + T(e)}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m, t, p\rangle \xrightarrow{b(e)} \langle c_1; \mathit{end}, m, t', p\rangle}$$

$$\frac{m(e) = 0 \qquad t' = t + T(e)}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m, t, p\rangle \xrightarrow{b(e)} \langle c_2; \mathit{end}, m, t', p\rangle}$$

$$\frac{t' = t + T(\mathit{end})}{\langle \mathit{end}, m, t, p\rangle \xrightarrow{f} \langle \mathit{stop}, m, t', p\rangle}$$

$$\frac{m(e) \neq 0 \qquad t' = t + T(e)}{\langle \texttt{while}\, e\, \texttt{do}\, c, m, t, p\rangle \xrightarrow{b(e)} \langle c; \mathit{end}; \texttt{while}\, e\, \texttt{do}\, c, m, t', p\rangle}$$

$$\frac{m(e) = 0 \qquad t' = t + T(e)}{\langle \texttt{while } e \texttt{ do } c, m, t, p\rangle \xrightarrow{b(e)} \langle \mathit{end}, m, t', p\rangle}$$

$$\frac{\begin{array}{c} t' = t + T(\texttt{setTimeout}(c, e)) \\ t'' = t + m(e) \qquad p' = \mathit{inSorted}((c, t''), p) \end{array}}{\langle \texttt{setTimeout}(c, e), m, t, p\rangle \xrightarrow{tin(t'', e)} \langle \mathit{stop}, m, t', p'\rangle}$$

$$\frac{t'' = \mathit{max}(t + T(\mathit{timeout}), t')}{\langle \mathit{stop}, m, t, (c, t').p\rangle \xrightarrow{\mathit{tout}} \langle c; \mathit{end}, m, t'', p\rangle}$$

Fig. 7. Semantics

External events record the effects of assignments, and are helpful for representing strong attackers that may observe changes in the public part of the memory. This accommodates smooth extension to the model of inputs and outputs by streams (which is an appropriate model of communication in deterministic programs [15]). Although we do not consider interactions with the outside world, we still use this formalism to minimize the impact of incorporating input and outputs to the language in future work.

Events Figure 7 presents the semantics for command configurations. Function T returns how many units of time takes to run a command or evaluate a given expression. We assume that $T(x)$ is constant for any variable x and that $T(e_1) = T(e_1')$ where e_1' is an expression obtaining from renaming any variable in e_1. Events s and $a(x, e)$ are triggered by commands `skip` and $x := e$, respectively. The semantic rules for these commands are self-explanatory. The rules for sequential composition are mostly standard except for requiring that $c_1 \neq \mathit{stop}$, which is related to timeouts and it will be explained in due course. Event $b(e)$, triggered by branching commands, indicates that the program branches on the expression e and is about to enter one of the branches. Expression e is a part of the event label so that if e involves secret data, the monitor will prevent any publicly observable behavior in the taken branch. The end command is added after the corresponding branch and triggers the event f, which informs the monitor that the block structure of a conditional has finished its execution. Similar to conditional branches, the semantic rule for loops also triggers event $b(e)$. When the loop's guard is non-zero, the command end executes after the body of the loop, i.e., `while` e `do` c is transformed into $c; \mathit{end};$ `while` e `do` c.

Timeouts The semantic rule for `setTimeout`(c, e) establishes a new timeout by inserting the pair (c, t'') into the list of timeouts p. The insertion of the pair preserves the order in p. To achieve that, we utilize $\mathit{inSorted}((c, t''), p)$ in the hypothesis of the rule. Time t'' indicates what is the time for command c to be run. In order to calculate that, the rule adds to the current time t the delay indicated by e ($t'' = t + m(e)$). The new current time is determined by considering how long it took to run the `setTimeout` instruction ($t' = t + T(\texttt{setTimeout}(c, e))$). Event $tin(t'', e)$ is triggered when setting timeouts. The reasons to include t'' and e as part of the event are explained in Section 7. When the list of timeouts is not empty, the rule for command stop triggers event tout, which indicates that a given timeout takes place. Observe that by triggering timeouts after reaching command stop implies that code snippets are only run after the source program or another code snippet finish its execution (as in JavaScript). The rule simply executes the first code snippet given in the list of timeouts p. We note $(c, t).p$ to the list of timeouts which first element is (c, t) and has a tail p. An end instruction is added after the code of the snippet. This addition is related to the monitor and it will be explained in Section 5. The new current time (t'') is determined by the maximum time between the time that it takes to trigger a timeout ($T(timeout)$) and the time that the first snippet is scheduled to run (t'). In this way, in the case that $t' > t + T(timeout)$, we do not need a rule in the semantics to wait until the current time is set to t'. On the other hand, when $t' < t + T(timeout)$, the snippet to be run at time t' is then run as soon as possible, i.e., at time $t + T(timeout)$.

We are now in a condition to explain the reason to require $c_1 \neq stop$ in one of the rules for sequential composition. In a standard semantics, the command $stop; c$ makes no progress since $stop$ cannot make any transition. However, in our semantics, if we did not require that $c_1 \neq stop$, it would be possible for $stop; c$ to make some progress since $stop$ commands could make transitions by executing code snippets associated with timeouts. Clearly, this behavior is not the expected one.

For simplicity, we present a semantics that models the scheduling strategies found in Explorer and Opera. To model timeouts in Firefox, it is enough to change the definition of *inSorted*, while more involved changes are needed to model the behavior of Konqueror, where the execution time of scripts affect the semantics for timeouts (again, one can argue that Konqueror's choice might be too anomalous to worry about). However, the principles of our enforcement mechanism remain unchanged independently of the browser's semantics.

clearTimeout, setInterval, and clearInterval The DOM API presents more primitives related to timeouts that we do not consider here. For example, primitive `clearTimeout` removes a given timeout. Although useful for programmers, considering this primitive would have complicated the language presented in Figure 7. The reason for that comes from the fact that the primitive operates on timeout handlers. More importantly, restricting the use of this primitive to be secure presents no novelty from the security point of view (see discussion in Section 5) and therefore, we omit it. Primitive `setInterval` works similarly to `setTimeout` except that it continues calling the function given as an argument until primitive `clearInterval` is executed. Thanks to the generality of the solution presented in the next section, `setInterval` can be handled in the same way as `setTimeout`, which again presents no novelty from the security point of view. We also omit `clearInterval` for the same reasons as for `clearTimeout`.

5. Enforcement

This section describes a runtime security enforcement mechanism for monitoring executions. A *monitor configuration* has the form $\langle st, \pi, \mu \rangle$ for a given stack of security levels st, a *security context list* π, and a *security context for setting timeouts* μ. For the moment, we ignore the purpose of the elements in the configuration (to be explained below). The monitor performs transitions of the form $\langle st, \pi, \mu \rangle \xrightarrow{\alpha} \langle st', \pi', \mu' \rangle$ where event α ranges over the events triggered by commands (see Figure 7). Intuitively, every time that a command triggers an event α, the monitor allows execution to proceed if it is also able to perform the labeled transition α. The rule for monitored execution in Figure 8 formalizes this intuition: every triggered event α is synchronized with the monitor. The monitor might disallow execution by stopping it (whenever it is unable to perform an α transition).

$$\frac{\langle c, m, t, p \rangle \xrightarrow{\alpha}_{\gamma} \langle c', m', t', p' \rangle \qquad \langle st, \pi, \mu \rangle \xrightarrow{\alpha} \langle st', \pi', \mu' \rangle}{\langle\, c, m, t, p \mid st, \pi, \mu \,\rangle \xrightarrow{\alpha}_{\gamma} \langle\, c', m', t', p' \mid st', \pi', \mu' \,\rangle}$$

Fig. 8. Monitored executions

$$\langle st, \pi, \mu \rangle \xrightarrow{s} \langle st, \pi, \mu \rangle$$

$$\frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle st, \pi, \mu \rangle \xrightarrow{a(x,e)} \langle st, \pi, \mu \rangle}$$

$$\frac{\pi' = lift(lev(e), \pi)}{\langle st, \pi, \mu \rangle \xrightarrow{b(e)} \langle lev(e).st, \pi', \mu \rangle}$$

$$\langle \ell.st, \pi, \mu \rangle \xrightarrow{f} \langle st, \pi, \mu \rangle$$

Fig. 9. Monitor rules I

$$\frac{\begin{array}{c}\ell' = lev(st) \sqcup lev(e) \\ \mu \sqsubseteq \ell' \qquad \pi' = inSorted((\ell', t), \pi)\end{array}}{\langle st, \pi, \mu \rangle \xrightarrow{tin(t,e)} \langle st, \pi', \mu \rangle}$$

$$\langle st, (\ell, t).\pi, \mu \rangle \xrightarrow{tout} \langle \ell.st, \pi, \mu \sqcup \ell \rangle$$

Fig. 10. Monitor rules II

Monitoring basic commands The semantics for the monitor is described in Figures 9 and 10. Figure 9 describes the behavior of the monitor for events associated to the part of the language unrelated to timeouts. For the moment, we ignore the parts of these rules marked with gray since they are related to timeouts, to be explained below. Event s, originated by `skip`, is always accepted without changing the monitor configuration. The stack of security levels st, which initially is empty (denoted by ϵ), keeps track of the dynamic *security context* [18], [26]: the security levels of the expressions appearing in the guards of branching commands (i.e., conditionals and loops). Intuitively, the security stack plays a similar role as program counters in security type systems [16][55]. Typing environment Γ associates every variable in the program with a security level. Since our approach is flow-insensitive, Γ is constant during the monitored execution of a program, and therefore we omit mentioning it in the monitor. It is also possible to provide a flow-sensitive monitor. However, in a purely dynamic enforcement mechanism, the flow sensitivity needs to be restricted to variables that are not part of commands that branch on secrets. For convenience, we only consider two security levels, low L and high H, as elements of a security lattice, where $L \sqsubseteq H$ and use the lattice join operator $\sqcup$ that returns the least upper bound over two given levels. Function $lev(e)$ returns the least upper bound of the security levels of variables encountered in expression e. Similarly, function $lev(st)$ returns the least upper bound of the security levels on the stack st. For the two-element lattice, the function returns H if there exists an element H in st, and L otherwise. Event $a(x, e)$, originated by $x := e$, is accepted without changes in the monitor state under two conditions. On the one hand, the security level of expression e is bounded from above by the security level of variable x ($lev(e) \sqsubseteq \Gamma(x)$), which prevents *explicit flows* of the form $l := h$ for a low variable l and a high variable h. On the other hand, the highest level of the security stack st is bounded from above by the security level of variable x ($lev(st) \sqsubseteq \Gamma(x)$), which prevents *implicit flows* [16] of the form $\texttt{if}\ h\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1$.

The rule for event $b(e)$ pushes the security level of e onto the security stack, which helps preventing implicit flows. For example, runs of the program $\texttt{if}\ h\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1$ are stopped before performing the assignments to l because the security stack contains H at the time of the assignments. For example, when running $l := 0$, the requirement of the rule for assignments $lev(st) \sqsubseteq \Gamma(x)$ does not hold since $lev(st) = H$ and $\Gamma(l) = L$. The stack structure avoids over-restrictive enforcement. For instance, runs of the program $(\texttt{if}\ h\ \texttt{then}\ h' := 0\ \texttt{else}\ h' := 1); l := 0$ are allowed since, by the time the assignment to l is reached, H has been removed from the stack in response to the event f, which is generated on exiting the scope of the conditional (recall Figure 6).

Monitoring timeouts To preserve confidentiality in the presence of timeouts, the monitor keeps track of more than just a simple stack of security levels. This additional information is represented in the monitor by a *security context list* π, and a *security context for setting timeouts* μ, which constitutes one of the novelties in our approach.

A *security context list*, which is initially set to empty, is a list of elements of the form (ℓ, t) for a given security level ℓ and time t. We only consider lists in an ascending order of time in the monitor. Intuitively, the monitor restricts a snippet to be run at time t to affect locations with confidentiality level ℓ or higher. To achieve that, the monitor rule for event $tout$ places ℓ into the security stack st. In Figure 7, the reason to include an end command in the semantic rule for $tout$ is just for removing ℓ from the security stack after the snippet finishes its execution. Intuitively, it is valid to think that every ℓ in the *security context list* is a program counter [16], [55] for a code snippet in a given timeout.

The *security context list* together with branching instructions play an important role to prevent internal timing attacks. For example, the attack presented in Figure 3 is no longer possible under monitored executions. Going back to the rules for branching commands in Figure 9, we observe that the security levels in the *security context list* π are lifted to the security level of the branching expression ($\pi' = \mathit{lift}(lev(e), \pi)$). In that way, when branching on `form.secret` in function `f`, code snippets `l0` and `leak` cannot perform observable events. As a result, the monitor stops the execution of function `l0` when assigning to the public variable `l` and thus preventing the leak. For similar reasons, the attacks in Figures 3 and 5 are also prevented. Although this mechanism avoids the attacks shown previously, the fact that code snippets can set timeouts still leaves open possibilities for leaks. In particular, code snippets might leak information by setting timeouts that affect public locations when the current time has been affected by secrets. To illustrate this point, we have the following example:

$$\begin{array}{l}\texttt{if}\ h\ \texttt{then}\ \texttt{setTimeout}(\texttt{skip};\texttt{skip};\texttt{skip}, 3)\\ \quad\ \texttt{else}\ \texttt{setTimeout}(\texttt{skip}, 3);\\ \texttt{setTimeout}(\texttt{setTimeout}(l := 0, 1), 2);\\ \texttt{setTimeout}(l := 1, 7);\end{array} \tag{1}$$

Considering that every step of the semantics takes a unit of time, we assume that the execution of the `then` and `else` branches take the same amount of time. Therefore, the current time at the first `setTimeout` instruction is the same regardless of the value of h. The snippets $\texttt{setTimeout}(l := 0, 1)$ and $l := 1$ are thus respectively associated to the times 5 and 11 when assuming a starting time 0. In the case that $h = 0$, the first snippet to be run is `skip`; `skip`; `skip` at time 6. After that, snippet $\texttt{setTimeout}(l := 0, 1)$ runs and sets the snippet $l := 0$ to be run at time 12. As a result, the final value of l is 0. On the other hand, if $h \neq 0$, the first snippet to be run is `skip` at time 6. Then, snippet $\texttt{setTimeout}(l := 0, 1)$ runs and sets $l := 0$ to be run at time 10. In this case, the final value of l is 1, which clearly produces a leak! The key observation here is that the time to run snippet $l := 0$ is influenced by running first other snippets that were created under a branch on secrets, which consequently affects the current time. With this vulnerability in mind, we introduce the *security context for setting timeouts* μ as a security level to restrict setting timeouts. Before explaining the purpose of μ, we firstly need to describe how `setTimeout` instructions are monitored. When setting timeouts (event tin), the monitor will restrict the locations affected by the code snippet to be as restrictive as the security context where the `setTimeout` instruction is executed as well as the confidentiality level of the indicated delay. This is achieved in the monitor by associating, to the recently created timeout, the security context $\ell' = lev(st) \sqcup lev(e)$ in π' ($\pi' = inSorted((\ell', t), \pi)$). Then, when a snippet associated with a security context ℓ is run, the *security context for setting timemouts* μ is updated by $\mu \sqcup \ell$ (see the monitor rule for event $tout$). In this way, μ is set to H as soon as a snippet created under a branching on secrets is executed. By restricting that $\mu \sqsubseteq \ell'$ when executing `setTimeout`, timeouts that set timeouts that affect public locations, like example (1), are not allowed to run till completion and therefore the vulnerability mentioned before is prevented.

clearTimeout, setInterval, and clearInterval To monitor `clearTimeout`, it is enough to restrict the elimination of timeouts created in contexts that are lower than the contexts where the elimination is carried out. In other words, the monitor needs to enforce that every time that a timeout associated to ℓ in π is eliminated, it is the case that $st \sqsubseteq \ell$. Since `setInterval` can be thought as a timeout that always set another timeout, the monitor rule for this instruction is the same as for `setTimeout`. Finally, the restrictions for `clearInterval` are the same as for `clearTimeout`.

6. Examples

This section evaluates how a full version of our monitor for JavaScript would behave for some common uses of timeouts. Timeouts are used for two common purposes: addressing possible failing XMLHTTP requests as well as for some interface issues as, for instance, animations. With this in mind, we show here running examples for these uses. We firstly need to describe the scenario considered for the examples. We assume that a user is connected to some origin O that involves some manipulation of secret information. For example, the user is connected to Gmail and a cookie in his (her) web browser contains the session ID for such connection, which is clearly secret data. Information that is sent to and received from the origin O is considered secret as well as everything that the user observes in his (her) display. We consider public sinks to any other origin different from the one where the user is connected. The following example, written using AJAX, shows the use of timeouts for addressing failing XMLHTTP requests.

```
[code lang="javascript"]
function callInProgress(xmlhttp) {
switch ( xmlhttp.readyState ) {
 case 1, 2, 3: return true; break;
 default     : return false; break;
} } [/code]
```

```
[code lang="javascript"]
var timeoutId = window.setTimeout(
function() { if (callInProgress(xmlhttp))
                { xmlhttp.abort(); }
}, 5000 ); [/code]
```

The code is self-explanatory. Observe that all the information handled by the code is secret. Consequently, the monitor would accept every execution of this script. Similarly, the next example would be also accepted by the monitor.

```
<script language="javascript">
var t;
function animBall(){
 t=setTimeout('animBall()', 80);
 moveRight(); doOtherThings(); }
</script>

<a href="javascript:animBall()">
Start Animation</a>
<a href="javascript:clearTimeout(t)">
Stop Animation</a>
```

In this case, the script produces an animation on the screen, i.e., a ball moving to the right.

At first glance, it seems reasonable to say that the techniques from our monitor is permissive enough to not "break" web pages due to the simply use of timeouts. However, more studies need to be performed in order to validate this claim.

7. Security

This section presents formal guarantees provided by the monitor. When showing the soundness of security

$(p_1, \pi_1, \mu_1, t_1) =_L (p_2, \pi_2, \mu_2, t_2) \Leftrightarrow$

$a)$ $time(p_1) = time(\pi_1)$

$b)$ $time(p_2) = time(\pi_2)$

$c)$ $snippet(filter_L(p_1, \pi_1)) = snippet(filter_L(p_2, \pi_2))$

$d)$ $\mu_1 = \mu_2 = L \wedge t_1 \geq t_2 \Rightarrow time(filter_L(p_1, \pi_1)) = time(filter_L(p_2, \pi_2)) + t_1 - t_2,$

$e)$ $\mu_1 = \mu_2 = L \wedge t_1 < t_2 \Rightarrow time(filter_L(p_2, \pi_2)) = time(filter_L(p_1, \pi_1)) + t_2 - t_1$

where $i \in \{1, \ldots, |filter_L(p_1, \pi_1)|\}$

Fig. 11. Low equivalence for timeouts

enforcement mechanisms, an attacker's view is often represented by an indistinguishability relation that describes what memories the attacker may or may not distinguish. The security soundness guarantees that program behaviors preserve memory indistinguishability: a program that starts with indistinguishable memories will not be able to distinguish between them over the course of the computation. For example, for a simple imperative language, such a relation consists on agreement of public values appearing in memories (e.g., [41]). In a timeout-based setting, however, we define an additional indistinguishability relation for timeouts (explained in Section 7.2). This relation does not enhance the attacker's view but rather help us to prove indistinguishability for memories. The core of our soundness results shows that our monitor preserves those relations for initial memories that agree on public values.

7.1. Security specification

We assume an attacker model, where the attacker can only observe externally observable events and the low part of the memories. Recall that we assume a strong attacker who can observe the low part of the initial memory and the effects of assignments to low variables. Events s, $b(e)$, f, $tin(t, e)$, and $tout$ are internal internal and hence externally unobservable. Event (x, v) is considered as high or low depending if x is a high or low variable, respectively. We denote a (possibly empty) sequence of monitored steps as $\xrightarrow{\vec{\alpha}}{}^*$, where $\vec{\alpha}$ is the list of triggered events. Similarly, we denote a (possibly empty) sequence of monitored steps, where the events are not relevant as $\longrightarrow{}^*$. We denote a single monitored step $\longrightarrow_\gamma$ as $\xrightarrow{L}_\gamma$ ($\xrightarrow{H}_\gamma$), when γ is a low (high) event. We denote a (possibly empty) sequence of monitored high steps $\xrightarrow{H}_\gamma$ as $\xrightarrow{H}{}^*_{\vec{\gamma}}$. The following predicate characterizes configurations that only trigger high events.

Definition 1 ($\Rightarrow_H$): $\langle\, c, m, t, p \mid st, \pi, \mu \,\rangle \Rightarrow_H$ iff for any sequence of steps such that $\langle\, c, m, t, p \mid st, \pi, \mu \,\rangle \longrightarrow^*_{\vec{\gamma}} \langle\, c', m', t', p' \mid st', \pi', \mu' \,\rangle$, we have that $\langle\, c, m, t, p \mid st, \pi, \mu \,\rangle \xrightarrow{H}{}^*_{\vec{\gamma}} \langle\, c', m', t', p' \mid st', \pi', \mu' \,\rangle$.

Observe that any monitored configuration that makes no progress satisfies Definition 1. Configurations that diverge without producing any public events or configuration that are stopped by the monitor satisfy Definition 1 as well.

We specify the security for programs via a noninterference-like condition [5]. Intuitively, if a program run is monitored, under some memory m, and produces a sequence of low events, then the same program under a low-equivalent memory m' ($m =_L m'$) will produce a prefix of that sequence. Formally:

Definition 2 (Security condition): Given a program c, the execution of c is secure if for any memories m_1 and m_2 such that $m_1 =_L m_2$, and $\langle\, c, m_1, 0, \epsilon \mid \epsilon, \epsilon, L \,\rangle \longrightarrow^*_{\vec{\gamma_1}} \langle\, c', m', t', p' \mid st', \pi', \mu' \,\rangle$, there exists c'', t'', p'', st'', π'', and μ'' such that $\langle\, c, m_2, 0, \epsilon \mid \epsilon, \epsilon, L \,\rangle \longrightarrow^*_{\vec{\gamma_2}} \langle\, c'', m'', t'', p'' \mid st'', \pi'', \mu'' \,\rangle$ where $|L(\vec{\gamma_2})| \leq |L(\vec{\gamma_1})|$ and

a) If $|L(\vec{\gamma_2})| = |L(\vec{\gamma_1})|$, then $L(\vec{\gamma_1}) = L(\vec{\gamma_2})$ and $m' =_L m''$.
b) If $|L(\vec{\gamma_2})| < |L(\vec{\gamma_1})|$, then $prefix(L(\vec{\gamma_2}), L(\vec{\gamma_1}))$ holds and $\langle\, c'', m'', t'', p'' \mid st'', \pi'', \mu'' \,\rangle \Rightarrow_H$.

Given a list of events $\vec{\gamma}$, $L(\vec{\gamma})$ projects out its low events. The number of events in $\vec{\gamma}$ is denoted by $|\vec{\gamma}|$. We also define predicate $prefix(\vec{x}, \vec{y})$ to hold when list $\vec{x}$ is a prefix of list $\vec{y}$.

7.2. Soundness

Indistinguishability relation In Figure 11 we present an indistinguishability relation for timeouts. It involves the list of timeouts p_1 and p_2, times t_1 and t_2, security context lists π_1 and π_2, and security contexts for setting timeouts μ_1 and μ_2. Functions $time()$ and $snippet()$ project out the time and snippet components of a given list, respectively. Function $filter_\ell(p, \pi)$ projects out elements of the form (c, t) in p which have associated a security context ℓ in π. The length of a list x is denoted by $|x|$. Requirements $a)$ and $b)$ establish that the list of timeouts and security contexts involved in the relation agree on the time components of their elements. Requirement $c)$ demands that the low-projections of snippets from p_1 and p_2 must be the same. Requirements $d)$ and $e)$ requires that there is a linear relation between the times for running snippets that might affect some public locations. These last two requirements help to guarantee that, when setting timeouts that affect public locations, they will be executed in the same order for any two executions that agree on public values.

Having defined the indistinguishability relation for timeouts and assuming the standard indistinguishability relation for memories (memories are indistinguishable by the attacker if they agree on the values of low variables), we then proceed to describe some lemmas that are needed to prove our main soundness result. We only present the main ideas and induction steps for the lemmas and theorems. The rest of the technical material appears in the full version of the paper [35].

We start by only considering configurations that are reachable from programs that do not include the commands end and $stop$, i.e., programs that are written by programmers. Formally:

Definition 3 ($\rightsquigarrow$): Given commands d and c such that d does not contain end and $stop$ instructions, predicate $d \rightsquigarrow \langle c, m, t, p \mid st, \pi, \mu \rangle$ holds iff there exists an initial memory m_i such that $\langle d, m_i, \epsilon, 0 \mid \epsilon, \epsilon, L \rangle \longrightarrow^* \langle c, m, t, p \mid st, \pi, \mu \rangle$.

The following lemma establishes some properties related to predicate $\rightsquigarrow$.

Lemma 1 (Reachability):

- If predicate $d \rightsquigarrow \langle c, m, t, p \mid st, \pi, \mu \rangle$ holds and $\langle c, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^* \langle c', m', t', p' \mid st', \pi', \mu' \rangle$, then $d \rightsquigarrow \langle c', m', t', p' \mid st', \pi', \mu' \rangle$ holds.
- If $d \rightsquigarrow \langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, t, p \mid st, \pi, \mu \rangle$ holds, then c_1 and c_2 contain no end and $stop$ instructions.
- If $d \rightsquigarrow \langle \texttt{while}\ e\ \texttt{do}\ c, m, t, p \mid st, \pi, \mu \rangle$ holds, then c contains no end and $stop$ instructions.
- If $d \rightsquigarrow \langle \texttt{setTimeout}(e, c), m, t, p \mid st, \pi, \mu \rangle$ holds, then c contains no end and $stop$ instructions.

Proof: By simple induction on $\longrightarrow^*$. □

From now on, unless we state it otherwise, we only consider configurations that are reachable from a source command d that contains no end and $stop$.

We firstly start by showing some lemmas related to the behavior of monitored executions. As stated in the following lemma, monitored executions can be composed sequentially. Having this feature implies that the monitor does not inspect commands to be run in the future in order to decide if the execution of an instruction is safe. Operation $\vec{x} +\!\!+\ \vec{y}$ concatenates the list of events $\vec{x}$ and $\vec{y}$.

Lemma 2 (Seq. composition of monitored executions): For any command c_2, we have that

i) given the non-empty sequence of steps $\langle c_1, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c_1', m', t', p' \mid st', \pi', \mu' \rangle$, where $c_1' \neq stop$ and no $tout$ events are triggered, then it holds $\langle c_1; c_2, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c_1'; c_2, m', t', p' \mid st', \pi', \mu' \rangle$.

ii) given the non-empty sequence of steps $\langle c_1, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle stop, m', t', p' \mid st', \pi', \mu' \rangle$ where no $tout$ events are triggered, then it holds $\langle c_1; c_2, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c_2, m', t', p' \mid st', \pi', \mu' \rangle$.

iii) given the non-empty sequence of steps $\langle c_1; c_2, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c', m', t', p' \mid st', \pi', \mu' \rangle$, then it holds that $c' = c_1'; c_2$ and $\langle c_1, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c_1', m', t', p' \mid st', \pi', \mu' \rangle$ where no $tout$ events are generated; or it is the case that there exists m'', t'', p'', st'', π'', and μ'' such that $\langle c_1, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma_1}} \langle stop, m'', t'', p'' \mid st'', \pi'', \mu'' \rangle$ where no $tout$ events are triggered and then $\langle c_2, m'', t'', p'' \mid st'', \pi'', \mu'' \rangle \longrightarrow^*_{\vec{\gamma_2}} \langle c', m', t', p' \mid st', \pi', \mu' \rangle$, where $\vec{\gamma} = \vec{\gamma_1} +\!\!+\ \vec{\gamma_2}$.

Proof: By simple induction on $\longrightarrow^*$. □

The following lemma states that the security stack in the monitor respects a stack structure.

Lemma 3 (Structure behavior of the security stack): Given a command c that contains no end instructions and given the semantics steps $\langle c, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle stop, m', t', p' \mid st', \pi', \mu' \rangle$, then $st = st'$.

Proof: By induction on $\longrightarrow^*$ and Lemma 2. □

The next lemma establishes that when the security stack in the monitor is high ($lev(st) = H$), then only high events are triggered and the low-equivalence relation between memories is preserved. Moreover, it is guaranteed that no timeouts that affect public locations are introduced.

Lemma 4 (No low events): Given a command c that contains no end instructions and given the steps $\langle c, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c', m', t', p' \mid st', \pi', \mu' \rangle$ where $lev(st) = H$, then it holds a) $m =_L m'$, b) $filter_L(p, \pi) = filter_L(p', \pi')$, and c) $\langle c, m, t, p \mid st, \pi, \mu \rangle \overset{H}{\longrightarrow}{}^*_{\vec{\gamma}} \langle c', m', t', p' \mid st', \pi', \mu' \rangle$.

Proof: By induction on $\longrightarrow^*$, Lemmas 2 and 3. □

Similarly to the previous lemma, loops with guards containing secrets do not trigger low events.

Lemma 5 (No low events in high loops): Given that $\langle \texttt{while}\ e\ \texttt{do}\ c, m, t, p \mid st, \pi, \mu \rangle \longrightarrow^*_{\vec{\gamma}} \langle c', m', t', p' \mid st', \pi', \mu' \rangle$ where $lev(e) = H$ and no $tout$ events are triggered, then it holds: a) $m =_L m'$, b) $filter_L(p', \pi') = \epsilon$, and $\langle \texttt{while}\ e\ \texttt{do}\ c, m, t, p \mid st, \pi, \mu \rangle \overset{H}{\longrightarrow}{}^*_{\vec{\gamma}}$

$\langle\, c', m', t', p' \mid st', \pi', \mu' \,\rangle$.

Proof: By induction on $\longrightarrow^*$, Lemmas 2 and 4. □

We now present one of the most important lemmas. Intuitively, the lemma states that given two execution that agree in public values, they will reach the same command and security stack before, and after, producing a low event.

Lemma 6 (Join point for low events): Given a command c, memories $m_1 =_L m_2$, $(p_1, \pi_1, \mu_1, t_1) =_L (p_2, \pi_2, \mu_2, t_2)$, and configurations

- $cfg_1 = \langle\, c, m_1, t_1, p_1 \mid st, \pi_1, \mu_1 \,\rangle$
- $cfg_1' = \langle\, c_1', m_1', t_1', p_1' \mid st_1', \pi_1', \mu_1' \,\rangle$
- $cfg_1'' = \langle\, c_1'', m_1'', t_1'', p_1'' \mid st_1'', \pi_1'', \mu_1'' \,\rangle$
- $cfg_2 = \langle\, c, m_2, t_2, p_2 \mid st, \pi_2, \mu_2 \,\rangle$
- $cfg_2' = \langle\, c_2', m_2', t_2', p_2' \mid st_2', \pi_2', \mu_2' \,\rangle$
- $cfg_2'' = \langle\, c_2'', m_2'', t_2'', p_2'' \mid st_2'', \pi_2'', \mu_2'' \,\rangle$

such that $cfg_1 \xrightarrow{H}{}^{*}_{\vec{\gamma_1}} cfg_1' \xrightarrow{L}_{\gamma_1} cfg_1''$ and $cfg_2 \xrightarrow{H}{}^{*}_{\vec{\gamma_2}} cfg_2' \xrightarrow{L}_{\gamma_2} cfg_2''$, then we have that $\gamma_1 = \gamma_2$, $c_1' = c_2'$, $c_1'' = c_2''$, $m_1' =_L m_2'$, $m_1'' =_L m_2''$, $(p_1', \pi_1', \mu_1', t_1') =_L (p_2', \pi_2', \mu_2', t_2')$, and $(p_1'', \pi_1'', \mu_1'', t_1'') =_L (p_2'', \pi_2'', \mu_2'', t_2'')$.

Proof: By induction on $\xrightarrow{H}{}^{*}_{\vec{\gamma_1}}$ and $\xrightarrow{H}{}^{*}_{\vec{\gamma_2}}$ and Lemmas 2, 3, 4, and 5. □

The following lemma establishes that two monitored executions that agree on public values will produce the same low events, stop, or diverge, in the latter case without producing any observable events for the attacker.

Lemma 7 (Backbone): Given a command c, memories $m_1 =_L m_2$, $(p_1, \pi_1, \mu_1, t_1) =_L (p_2, \pi_2, \mu_2, t_2)$, and configurations

- $cfg_1 = \langle\, c, m_1, t_1, p_1 \mid st, \pi_1, \mu_1 \,\rangle$
- $cfg_1' = \langle\, c_1', m_1', t_1', p_1' \mid st_1', \pi_1', \mu_1' \,\rangle$
- $cfg_1'' = \langle\, c_1'', m_1'', t_1'', p_1'' \mid st_1'', \pi_1'', \mu_1'' \,\rangle$
- $cfg_2 = \langle\, c, m_2, t_2, p_2 \mid st, \pi_2, \mu_2 \,\rangle$
- $cfg_2' = \langle\, c_1', m_2', t_2', p_2' \mid st_2', \pi_2', \mu_2' \,\rangle$
- $cfg_2'' = \langle\, c_1'', m_2'', t_2'', p_2'' \mid st_2'', \pi_2'', \mu_2'' \,\rangle$

such that $cfg_1 \xrightarrow{H}{}^{*}_{\vec{\gamma_1}} cfg_1' \xrightarrow{L}_{\gamma_1} cfg_1''$, then it holds that either $cfg_2 \Rightarrow_H$ or $cfg_2 \xrightarrow{H}{}^{*}_{\vec{\gamma_2}} cfg_2' \xrightarrow{L}_{\gamma_2} cfg_2''$ where $m_1' =_L m_2'$, $m_1'' =_L m_2''$, $(p_1', \pi_1', \mu_1', t_1') =_L (p_2', \pi_2', \mu_2', t_2')$, and $(p_1'', \pi_1'', \mu_1'', t_1'') =_L (p_2'', \pi_2'', \mu_2'', t_2'')$.

Proof: By case analysis on the veracity of $\langle\, c, m_2, t_2, p_2 \mid st, \pi_2, \mu_2 \,\rangle \Rightarrow_H$ and then applying Lemma 6. □

We present the main soundness result.

Theorem 1 (Soundness): For any memory m and command c, the execution of c starting at the configuration $\langle c, m, 0, \epsilon \mid \epsilon, \epsilon, L\rangle$ is secure according to Definition 2.

Proof: By induction on the number of low events and application of Lemma 7. □

Finally, the following corollary relates Definition 2 with a batch-job style termination-insensitive security condition (e.g., [55]).

Corollary 1 (Batch-job soundness): Given a program c, memories m_1 and m_2 such that $m_1 =_L m_2$ and two terminating monitored runs : $\langle\, c, m_1, 0, \epsilon \mid \epsilon, \epsilon, L \,\rangle \longrightarrow^*_{\vec{\gamma_1}} \langle\, stop, m_1', t_1', \epsilon \mid \epsilon, \epsilon, \mu_1 \,\rangle$ and $\langle\, c, m_2, 0, \epsilon \mid \epsilon, \epsilon, L \,\rangle \longrightarrow^*_{\vec{\gamma_2}} \langle\, stop, m_2', t_2', \epsilon \mid \epsilon, \epsilon, \mu_2 \,\rangle$, then it holds that $m_1' =_L m_2'$ and $L(\vec{\gamma_1}) = L(\vec{\gamma_2})$.

Proof: By Theorem 1 and Definition 2. □

8. Related work

We refer to Sabelfeld and Myers' survey [41] for general background on language-based information-flow security. We formalize the reflections in Section 2 and prove for a simple language with output that a purely dynamic information-flow monitor is more permissive than a Denning-style static information-flow analysis, while both the monitor and the static analysis guarantee the same security property: termination-insensitive noninterference [42]. For similarly-spirited work on dynamic information-flow control for systems with declassification and DOM tree operations, we refer to work by Askarov and Sabelfeld [8] and Russo et al. [37], respectively. The compositional nature of the underlying monitor enables a natural combination of these enforcement techniques (which do not address timing issues) with the one presented here. In the rest of this section, we focus on related work involving timing attacks.

Internal timing There is a body of work on statically preventing internal timing attacks, sometimes in combination with runtime mechanisms, for multi-threaded programs. As discussed previously, our technique offers more permissiveness compared to static methods and yet guarantees termination-insensitive security.

In earlier work, we have explored the interaction between the threads and the scheduler [33] in order to control internal timing leaks. The interaction is modeled by *hide* and *unhide* primitives that communicate to the scheduler whether a thread's timing behavior should be "hidden". We have shown that this mechanism is sufficient for security in addition to static checks for explicit and implicit flows. In collaboration with Barthe and Rezk, we have extended this approach to low-level languages [10]. In the latter work, there is no need for explicit hide/unhide primitives because scheduler is driven by the security context. If a thread is inside of a conditional with a high guard, then it executes in a high security environment and thus its timing behavior is automatically hidden from threads that run in a low security environment.

Approaches by Volpano and Smith [48], [54], [46], [47] to handling internal timing rely on $\texttt{protect}(c)$ which, by definition, hides the internal timing of command c. It is not clear how to implement $\texttt{protect}()$ without modifying the scheduler (unless the scheduler is cooperative [34], [50]). It is possible to prevent internal timing leaks by spawning dedicated threads for computations that involve secrets and carefully synchronizing the resulting threads [32], although with high synchronization costs. Yet other approaches prevent internal timing leaks in code by disallowing races on public data [56], [22], [49]. However, they reject such innocent programs as $l := 0 \parallel l := 1$, where $\parallel$ is a parallel composition operator, and l is a public variable. Still other approaches prevent internal timing by disallowing low assignments after high branching [13], [7]. Less related work [6], [43], [39], [40], [25] considers external timing, where an attacker can use a stopwatch to measure computation time. A price paid for security against this kind of more powerful attackers is disallowing loops to branch on secrets.

Further afield, different flavors of possibilistic noninterference have been explored in process-calculus settings [20], [19], [38], [21], [31], but without considering the impact of scheduling. Recently, van der Meyden and Zhang [51] have investigated how the choice of a scheduler can affect security definitions in an abstract automata-based setting.

Timing attacks in web applications Johns [24] reports on the following recent work in this area. Felten and Schneider [17] demonstrate a timing attack that allows discovering whether the user has recently visited a certain web page. The attack is based on time difference between retrieving a cached and non-cached web object. Jackson et al. [23] propose a protection mechanism against this type of attacks by forcing cache and history information to be inaccessible by scripts of a different origin.

Bortz et al. [11] extend the technique to non-cacheable web objects. They introduce *cross-site timing attacks*, where they exploit the difference between the time it takes to create an object by the victim server. This difference can be substantial: for example, the time to look up a static object and to respond to a request that involves database requests are likely to be significantly different. To resolve the problem, they propose a server-side mechanism that ensures that each request is processed by the server in constant time.

Both of these attacks are external timing attacks; they involve time measurement operations and have a lower bandwidth, compared to internal timing attacks. While external timing attacks are outside the scope of this paper, the most straightforward way to extend our mechanism with external time operations would be to treat the result of accessing a clock secret [48].

9. Conclusions

We have presented a runtime enforcement mechanism for security information flow in applications with timeout primitives. We show that internal timing leaks can be secured by a simple runtime mechanism. We believe this is a good fit for web application security, where it is important to break away from the traditional static approaches that are often over-restrictive to handle dynamically generated code. On the implementation side, we believe that is not difficult to include the monitor as a plug-in or as part of the web browser. Then, the browser creates an instance of the monitor for every connection to a website that involves secrets (i.e., cookies and session ids).

At the time of writing, there are ongoing discussions on incorporating concurrency support in the DOM for Firefox 3.1 [44]. In fact, threads can be already emulated with help of *generators* [3] already in Firefox 2.0 [1]. We expect that our solution for timeouts can be adopted to address concurrency. Threads can be tracked similarly to the commands in the timeout queue: whenever the main thread branches on secret data, the other live threads are “tainted”, preventing them from assigning to public variables. Also, when a tainted thread executes, threads to be created in the future are also tainted. We expect that this simple mechanism, together with preventing thread creation by threads that were tainted while in the thread pool, is sufficient to guarantee information-flow security.

This work is a part of a larger effort (e.g., [8], [37]) on hybrid approaches to securing web applications. Current and future work is focused on integrating the enforcement mechanism described here with ones that address orthogonal issues but share the basic philosophy. The long-term goal is to have both a formal foundation and a practical implementation for controlling information flow in a realistic language like JavaScript and address a substantial part of DOM API. To achieve that, it is necessary, among other challenges, to deal with different covert channels present in the language. As the two lines of work evolve, we plan to perform case studies of expressiveness and performance, which would be crucial for evaluating our approach.

Acknowledgments We thank our colleagues in the ProSec group at Chalmers and the anonymous reviewers for the useful comments. This work was funded by the Swedish research agencies SSF and VR.

References

[1] Threading in JavaScript 1.7. http://www.neilmix.com/2007/02/07/threading-in-javascript-17, 2007.

[2] Mozilla developer center: window.setTimeout. https://developer.mozilla.org/en/DOM/window.setTimeout, 2008.

[3] New in JavaScript 1.7: Iterators. https://developer.mozilla.org/en/New_in_JavaScript_1.7-#Generators, 2008.

[4] XSS Attacks Information. http://www.xssed.com, 2008.

[5] A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, Oct. 2008.

[6] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[7] A. Almeida Matos. *Typing secure information flow: declassification and mobility*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2006.

[8] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[9] J. Barnes and J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[10] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symp. on Research in Computer Security*, volume 4734 of *LNCS*, pages 2–18. Springer-Verlag, Sept. 2007.

[11] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applications. In *Proc. International Conference on World Wide Web*, pages 621–628, 2007.

[12] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*, pages 382–395. Springer-Verlag, July 2001.

[13] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[14] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[15] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, Oct. 2008.

[16] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[17] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[18] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[19] R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.

[20] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.

[21] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, Jan. 2002.

[22] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

[23] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting browser state from web privacy attacks. In *Proc. International Conference on World Wide Web*, 2006.

[24] M. Johns. On JavaScript malware and related threats. *Journal in Computer Virology*, 4(3):161–178, Aug. 2008.

[25] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*, pages 47–62. Springer-Verlag, July 2006.

[26] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.

[27] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[28] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[29] Netscape. Using data tainting for security. http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm, 2006.

[30] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[31] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[32] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference (ASIAN'06)*, LNCS. Springer-Verlag, 2007.

[33] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[34] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2006.

[35] A. Russo and A. Sabelfeld. Securing Timeout Instructions in Web Applications. Full version. http://www.cse.chalmers.se/~russo/, 2009.

[36] A. Russo and A. Sabelfeld. Timeout Attacks. http://www.cse.chalmers.se/~russo/websecurity/, 2009.

[37] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures, Apr. 2009. Draft.

[38] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[39] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[40] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[41] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[42] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[43] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[44] E. Shepherd. Using dom workers. https://developer.mozilla.org/En/Using_DOM_workers, 2009.

[45] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

[46] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[47] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.

[48] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[49] T. Terauchi. A type system for observational determinism. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[50] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium*, July 2007.

[51] R. van der Meyden and C. Zhang. Information flow in systems with schedulers. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[52] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.

[53] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[54] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[55] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[56] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.