# Disclosure: Improving Performance and Security of Web App Migration in Liquid Computing

Jae-Yun Kim* and Soo-Mook Moon

*Seoul National University, Seoul, Republic of Korea*
*E-mail: jaeyun.kim@snu.ac.kr; smoon@snu.ac.kr*
**Corresponding Author*

Received 01 October 2022; Accepted 14 November 2022;
Publication 14 April 2023

## Abstract

*Web app migration* refers to capturing a snapshot of the execution state of a web app on a device and restoring it on another device to continue its execution for cross-device *liquid computing*. Although web apps are relatively easy to migrate due to their high portability, there is a JavaScript language feature called *closure* that complicates the migration since it requires migrating the variable states of already-finished outer functions. One approach to web app migration is to instrument the source code to trace the closure variables, yet this often suffers from performance slowdown, especially for multiple migrations. In this paper, we propose a new instrumentation-based technique called *Disclosure*, which moves the declarations of closure variables to a managed data structure and replaces the closure variables with the corresponding references to the data structure. This technique can improve runtime performance while enhancing security. We evaluated our work with eight *Octane* benchmarks and four real web apps. The runtime performance penalty due to Disclosure is 0–15%, which is a significant improvement over the results of the latest instrumentation-based work that supports similar deep

*Journal of Web Engineering, Vol. 22_1,* 79–104.
doi: 10.13052/jwe1540-9589.2215

closures and multiple migrations to Disclosure. Furthermore, real web apps are demonstrated to migrate seamlessly, even multiple times. Finally, Disclosure can hide data from exposure during migration with a secure migration technique using data encryption.

**Keywords:** Web app migration, closure, code instrumentation, liquid computing, multiple migrations, secure migration.

## 1 Introduction

Following the considerable advancement of web technology, *JavaScript* has become one of the most popular programming languages used today [1]. In addition, the web browser has become the dominant platform for various technological environments, such as PCs, smartphones, smart TVs, and IoT devices. Since web applications can run anywhere regardless of CPU or operating system, *app migration* has readily emerged. App migration captures a snapshot of an application state during the middle of execution and restores it to another device to continue the execution [2–6]. App migration can offer a novel user experience; for example, a game application running on a smartphone could be handed over to a smart TV for continued execution on a larger screen, then to another device as the user moves elsewhere. It is a form of *liquid computing* [7]. For a manufacturer that can provide a uniform web platform for its diverse smart device products (e.g., Samsung Tizen or LG WebOS), the difficulty of liquid computing caused by the differences in the device specification or the web browser context is more easily surmountable, making liquid computing more feasible.

A snapshot captures the state of the objects in the heap memory along with other states in the *JavaScript Runtime* (JR). However, there are some challenges with capturing them. *Closure*, a language feature inherent to JavaScript, is a function containing *free variables*, which are used inside the function but declared outside the function scope. If a free variable is still alive (referenced) after the outer function is terminated, it becomes a closure variable and is stored as a closure object in the heap memory. Closure variables are inaccessible from the *execution context* (stack frame in JavaScript) or the global *window* object, thus demanding a new strategy to capture them. In addition, JR includes *Document Object Model (DOM)*, *XMLHttpRequest (AJAX)*, and *timeout methods*. *DOM objects* are stored in a tree structure and are accessible via Web APIs provided by the browser, so we can capture the DOM state by traversing the *DOM tree*. Meanwhile, timeout methods

employ JR's timer objects to schedule the execution of callback functions; however, accessing the registered timer's state is impossible since there is no Web API. Finally, XMLHttpRequest is used to interact with servers and is not typically within the scope of app migration since app migration only targets applications that can run standalone. Thus, the app migration technique has focused on capturing closures and timers, and two approaches have been proposed.

One approach is to instrument the source code of a web application to trace the closures within the hierarchy of the *scope tree*, a combination of scope chains, by inserting a *mirroring statement* [3, 6] under any statement that includes the closure variables. Such mirroring statements can trace the closure variables with their position in the scope tree, but this increases the program size and, in turn, the running time. Timers are handled by the *wrapper functions*, which wrap timeout methods to record the arguments passed when an event is registered, allowing rescheduling them from the moment of migration. Overall, this approach successfully captures the snapshot but suffers from severe performance degradation due to the overhead of the mirroring statements. A recent work improved the performance by taking a snapshot lazily [8], yet it does not allow for multiple migrations, thus limiting the liquidity of the cross-device user experience.

The other approach is to add new APIs to JR, which is a method of directly accessing the closures and timers via modifying web browsers [2, 4, 5]. This approach leaves the source code intact and thus does not affect runtime performance. However, it weakens the security of applications since JavaScript developers typically implement data encapsulation through closures. Therefore, such APIs are not generally welcomed by browser vendors, and users must use a custom browser for migration, lowering the portability.

This paper proposes a new instrumentation-based technique called *Disclosure*, which moves declarations of the closure variables to a managed data structure called a *disclosure table* and replaces the closure variables with the references to the corresponding elements within the table. In this way, Disclosure obviates mirroring statements and significantly reduces runtime overhead, maintaining almost the same performance as the original program. Moreover, we can implement the disclosure table itself *as a closure*, which can keep the closure variables from being revealed, enhancing security further than in previous works. We capture the timers using the wrapper functions as done previously, yet store them in the disclosure table as well. Disclosure can fully capture the DOM tree, unlike previous instrumentation-based works. A user can consequently take a snapshot by copying the DOM tree, the

objects in the heap memory, and the disclosure table. This snapshot, written in JavaScript, is a full-fledged web application by itself, so we can simply run it on the browser of the target device to continue execution with its current display. We make the following contributions:

(1) We propose a novel instrumentation-based migration technique for closures that can keep the instrumented program and the snapshot program from being slowed down seriously while allowing multiple migrations.
(2) Our snapshot can preserve security for closure variables, possibly enhanced with cryptographic methods.
(3) Disclosure can migrate a whole execution state of a web app, including JavaScript and DOM tree, unlike most previous instrumentation-based works.
(4) Our evaluation with Octane benchmarks shows a tangible performance benefit, while real web apps are shown to migrate seamlessly, multiple times.
(5) We propose secure migration to hide snapshot data from being exposed during migration.

The rest of this paper is as follows. We provide a background on JavaScript runtime in Section 2 and explain the challenges for web app migration in Section 3. We review the previous approaches in Section 4, then describe the technical details of Disclosure in Section 5. Section 6 presents our evaluation results and also compares them with the previous work. Section 7 depicts secure migration, a way to hide the execution state during migration. Section 8 describes related work with real-life examples followed by the conclusion in Section 9.

## 2 Background of JavaScript Runtime

The JavaScript Runtime comprises a JavaScript engine and other runtime components. The JavaScript engine is composed of the *call stack* (execution context stack) and *heap memory*, and other components include the event queue, web APIs, and event loop. When a JavaScript application is loaded, a global execution context is first pushed to the call stack. If a new function is invoked, a new execution context is generated, referencing the global context at the top of the stack. This process is repeated so that each execution context references the outer (previous) execution context, and this chain-structured execution of contexts is called the *scope chain*. During this process, the developers register *event handlers* by using *event listeners* to address asynchronous

events triggered by a button click, timers, etc. A triggered event is pushed to the event queue with the registered event handler. When a function is terminated, conversely, its execution context is removed from the top of the stack. Eventually, the call stack will be empty after all functions and the code in the global scope have been executed. Then, the event loop fetches an event and the event handler from the event queue to execute it on a new execution context assigned to the call stack. This is the method by which JavaScript handles asynchronous tasks. Since the JavaScript engine is single-threaded, it runs only one event at a time and cannot execute another until the current event is terminated. Considering how the browser works, the simplest way to implement app migration is to make it an event since the call stack would be empty when the migration event is fetched from the event queue. This strategy has the advantage of not needing to capture the state of the call stack. Therefore, we implemented the migration task as an event so that a developer or user can call it asynchronously via a console or a browser extension.

## 3 Challenges for Web App Migration

JavaScript functions are objects (first-class functions), which allow for defining a function within another function scope, thereby creating a nested function structure. JavaScript enables an inner function to access the *free variables* defined at one of the outer functions. Those free variables are accessible even after the lifecycle of the outer function is terminated since the inner function is established as a closure containing the lexical environment of the outer function with the free (closure) variables. However, the environment of the terminated function is not accessible from the outside, so developers use this feature to implement data encapsulation in JavaScript. The environment is the variable-value mapping of the current scope chain, and each execution context has an internal property called *scope* used to reference the previous execution context. Listing 1 provides a code example for scope chain and closures. The variable *count* (line 2) is used in the inner function (lines 3–6). Since it is a local variable of the outer function *CreateCounter()*, it is removed from the stack once the outer function terminates. However, it is not eliminated but rather is saved as a closure variable in the JavaScript heap since it is still used by the returned inner function. We can depict the scope chain and closure variable *count*.

Timeout methods utilize JR's timer to schedule the execution of event handlers that are pushed to the event queue when the timer expires. However, since web browsers do not provide Web APIs for accessing the timer,

```
1  function CreateCounter() {
2      var count = 0
3      return function() {
4          count += 1
5          console.log(count)
6      }
7  }
8  var myCounter = CreateCounter()
9
10 setInterval(function () {
11     myCounter()
12 }, 1000)
```

**Listing 1** An example JavaScript code with a closure variable.

capturing their states is challenging for app migration. Listing 1 describes an example of a timeout method, *setInterval* (line 10). This method repeatedly registers a new timer event, which has an anonymous callback function (event handler, lines 10–12) that calls *myCounter* (line 11). Therefore, the event will be pushed to the event queue every second (line 12). When a migration event is pulled from the event queue and executed, the migration process must capture the state of the timer to register any pending events and repeated ones thereafter, following migration.

## 4 Previous Approaches

Two different approaches to web app migration are available for handling closures and timers. One approach is to instrument the source code of the target application statically, while the other is to modify the web browser to provide new APIs.

*Imagen* [3] and *ThingsMigrate* [6] instrument the source code by inserting mirroring statements as depicted in Listing 2. Since these are similar, we explain based on ThingsMigrate, which can migrate deep closures, unlike Imagen. ThingsMigrate creates scope objects (lines 1, 3) for every execution context, including the relevant hierarchical information of each scope chain to establish a scope tree. It then allocates all variables as inherent properties of the corresponding scope objects (lines 5, 12, 15). If a statement affects any of the variables, a mirroring statement is inserted below (lines 8, 17) to copy the updated value. Moreover, it wraps each timeout method to trace an event handler and timer argument (lines 19–21). Thus, it replaces *setInterval()* with the wrapper function *ThingsMigrate.setInterval()* to record the event handler *myCounter*, the time interval (1000 ms), and the remaining time before registering the subsequent event.

The migration process produces a snapshot that captures all the objects in the heap memory and the timers. This snapshot is generated in the form of JavaScript code so it can be executed on any device with any web browser, as depicted in Listing 3. If a migration event occurs 3.75 s after the app loads, the value of the closure variable *count* is 3, and the remaining system time for the next event is 0.25 s. ThingsMigrate generates a snapshot in the form of an *immediately-invoked function* to restore the closures without any side effects (lines 2–9). Subsequently, the timers are restored with the wrapper function again to allow for capturing them for multiple migrations (lines 11–13).

The second approach is to add new APIs to web browsers with the aim of retrieving information about closures and timers [2,4,5]. This strategy leaves the source code intact, allowing users to capture and migrate any applications to other devices without the need for instrumentation. In addition, the runtime performance of the application is identical to that of the original.

Both approaches have successfully implemented app migration technology; however, we found issues with both methods. The former approach, which utilizes code instrumentation, results in a serious performance slowdown to runtime due to the mirroring statements needed to trace the scope tree (lines 8, 17 in Listing 2). Although *FlashFreeze* [8] solves this problem by ignoring scope objects and tracing only the closures, it cannot support the multiple migrations required for liquid computing (its performance is similar to our previous version's, presented in a work-in-progress report). Furthermore, the former approaches support only the migration of the JavaScript state, not the DOM tree (only Imagen partially migrates the DOM objects), and thus are not suitable for migrating web applications. On the other hand, the latter approach cannot conceal private information, as anyone can look into the closure variables and timer states using the new APIs. Moreover, users should employ a customized web browser for migration, which reduces the portability.

## 5 The Disclosure Approach

Disclosure is a new approach to solving the issues of the previous methods. It extends the code instrumentation scheme yet obviates the mirroring statements to improve performance. We capture an application's scope tree by traversing from the *window* object instead of using a mirroring statement. Since a closure cannot be accessed from the *window* object, the instrumented code moves the declarations of the closure variables to a managed data structure called a *disclosure table* and replaces the variables with the references

```
var global = new Scope("global")
function CreateCounter() {
    var createcounter = new Scope(global, "CreateCounter")
    var count = 0
    createcounter.addVar("count", count)
    var anon1 = function() {
        count += 1
        createcounter.setVar("count", count)
        console.log(count)
        return count
    }
    createcounter.addFunction("anon1", anon1)
    return anon1
}
global.addFunction("CreateCounter", CreateCounter)
var myCounter = CreateCounter()
CreateCounter.setVar("myCounter", myCounter)

ThingsMigrate.setInterval(function() {
    myCounter()
}, 1000)
```

**Listing 2** The instrumented code generated by *ThingsMigrate* for Listing 1.

```
/* The instrumented code (Listing 2) comes here */
(function() {
    function CreateCounter() {
        var count = 3
        var anon1 = function() { ... }
        ThingsMigrate.addFunction("Global/CreateCounter/anon1",anon1)
        return anon1
    }
})()

ThingsMigrate.setInterval(ThingsMigrate.
    findFunction("Global/CreateCounter/anon1"),
    1000, 250)
```

**Listing 3** Snapshot code serialized by *ThingsMigrate*, 3.75 s after app loading of Listing 2.

to the corresponding elements within the table. Serializing the timers or generating a snapshot file works similarly as in previous works, but handling the DOM tree is done differently.

Disclosure consists of three phases: *the instrumentation phase, execution phase, and migration phase*. The instrumentation phase transforms the source code by changing the closure variables using the disclosure table, wrapping the timeout methods, and generating the conversion code for the DOM objects. Then, the execution phase utilizes a runtime library with the disclosure table to store the structures and values of the closures. Finally, the migration phase captures the DOM tree, heap objects, timers, and disclosure

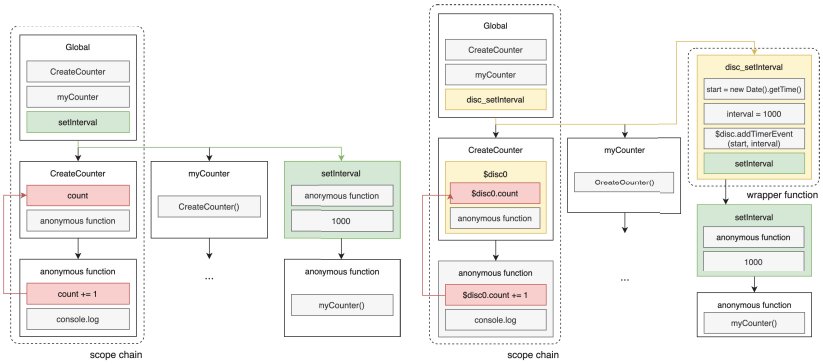
table to produce a snapshot file. We also explain a side effect, memory leak, and a solution, WeakMap.

## 5.1 Instrumentation Phase

The instrumentation phase finds free variables and determines those likely to become closure variables at runtime. This phase utilizes an *Abstract Syntax Tree* (AST) to emulate scope chains to discern the closure variables, generates the code to move their definitions to the disclosure table, and replaces them with the corresponding references to the elements in the disclosure table. In addition, this phase converts each timeout method into a wrapper function to copy the arguments to trace the timers and re-register them after migration. This phase consists of three sub-phases: *AST Generation*, *AST Traversal*, and *Code Generation*.

The AST is a fundamental data structure for code analysis, and we use it to determine whether a free variable is a closure variable. Since the AST's hierarchical structure represents function inclusion orders, each path is likely to be instantiated to a scope chain at runtime. Figure 1(a) provides an AST-like representation of Listing 1, and we can observe a scope chain on the left side composed of *Global*, *CreateCounter function*, and *anonymous function* scopes. Meanwhile, on the right side, we can find the timeout method, *setInterval*, defined in the global scope. It displays the event handler that will be pushed to the event queue every 1000 ms.

As mentioned previously, we can infer a scope chain using an AST path. Therefore, we traverse the AST in a Depth First Search (DFS) order to identify the closure variables with a virtual stack to emulate the scope chain. Our *traverser* begins from the root node, the *window* object, and when it encounters *VariableDeclaration*, an AST node, it pushes the declared variable to the virtual stack with its expected execution context. If the traverser encounters *ExpressionStatement*, it determines whether the variable is a closure variable by using the virtual stack. For example, in Figure 1(a), if the traverser encounters *FunctionExpression* that includes the *count += 1* statement (red box), it can identify that the variable *count* is a closure variable because it is not declared within the anonymous function scope. Then, it explores the virtual stack to determine in which function scope the variable count is defined. In this case, it finds that the variable *count* is declared in the *CreateCounter* function scope, meaning it would be a closure variable at runtime. However, JavaScript allows for assigning a value to an undeclared variable, which is regarded as being declared in the global scope,




(a) AST-like representation of scope chains for identifying closure variables. (b) Updated representation to save closure variables and event handlers.

**Figure 1** Scope chains with closure variables.

so Disclosure does not treat those undeclared variables as closure variables but rather inserts them into the global scope for further analysis. Since those global variables would be accessible from the *window* object, we can infer they are not closure variables. Finding the timeout methods is simpler because they are predefined at the global scope (e.g., we can identify *setInterval*, as illustrated in Figure 1(a) (green box), by singling out the character string "setInterval").

When the traverser encounters a closure variable, it inserts a *declaration statement* that generates a scope object into the disclosure table at the top of the function scope where the closure variable is declared. Simultaneously, it replaces each closure variable with a property of the scope object in the disclosure table. For example, the instrumented version of the original source code in Listing 1 can be depicted in Listing 4. Since the variable *count* is a closure variable (line 6), the traverser inserts a statement that produces the scope object *$disc0* at the top of the function scope (line 2). The expression *$disc.create()* creates a scope object within the disclosure table and returns the reference. Subsequently, the traverser inserts a statement to record the index of the scope object *$disc0* using the global reference counter *$ref_counter*. Then, each closure variable is converted into a property of the new scope object (lines 4, 6, 7). The inner function to be returned and serve as the closure is declared as the property of the scope object for subsequent closure reconstruction (line 9). When the timer method *setInterval* is met, it is replaced by a predefined wrapper function *disc_setInterval* (line 13) to copy the argument and system time. The AST is then updated as in Figure 1(b).

```
1  function CreateCounter() {
2      var $disc0 = $disc.create()
3      $disc0.$scopes.$disc0 = $disc0.$ref_index
4      $disc0.count = 0
5      $anon0 = function () {
6          $disc0.count += 1
7          console.log($disc0.count)
8      }
9      return $disc0.$ret_func = $anon0
10 }
11 var myCounter = CreateCounter()
12
13 disc_setInterval(function () {
14     myCounter()
15 }, 1000)
```

**Listing 4** The instrumented code generated by Disclosure for Listing 1.

```
1 create: function() {
2     var obj = new Object()
3     obj.$ref_index = disc_table.length
4     obj.$scopes = {}
5     disc_table.push(obj)
6     return obj
7 }
```

**Listing 5** How *create* function works.

## 5.2 Execution Phase

The execution phase uses a runtime library that maintains the disclosure table. It declares wrapper functions for timeout methods within the global scope. The runtime library is allocated to the global scope, accessible with *$disc*, which has the *create* function as presented in Listing 5. The function generates a new scope object (line 2) and records the index of the scope object to the property named $ref_index (line 3). Moreover, it has the *$scopes* object, which maps each scope object to the corresponding index (line 4). Finally, it pushes the new scope object to the disclosure table and returns it (lines 5, 6).

If 3.75 s have passed since the code in Listing 4 was loaded and executed, the first scope object in the disclosure table can be depicted as in Listing 6. It maintains the closure variable *count*, whose value is 3 (line 3), in addition to the index of itself with the variable name {*$disc0: 0*} (line 4). Finally, it records the object literal that will serve as a closure for subsequent closure reconstruction (lines 5–8). The state of the AST is depicted in Figure 2, in which the variable *count* is referred to as a property of the scope object *$disc[0]* in the disclosure table.

```
/* $reference_table[0] */
{
    count: 3,
    $scopes: {$disc0: 0},
    $ret_func: "function () {
      $disc0[\"count\"] += 1
      console.log($disc0[\"count\"])
    }"
}
```

**Listing 6** An element state of the disclosure table.

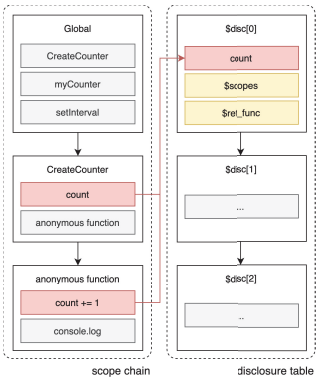



**Figure 2** Scope chain and transferred closure of Listing 4.

## 5.3 Migration Phase

The runtime library has a *serialize* function, which pushes an event handler to save the DOM objects, heap objects, and disclosure table. Users can transfer the snapshot to another device to restore it and continue its execution. Since the snapshot code runs similarly to the instrumented code, users can again capture a snapshot during the execution of the snapshot code.

User interactions in a web application are typically conducted via DOM objects, such as buttons, after the global context is terminated and the call stack becomes empty. Therefore, the serialization event does not need to capture the call stack. It first generates a snapshot file containing the DOM objects and the instrumented code and then traverses the heap memory to capture the global variables. Subsequently, it serializes the disclosure table to append the reconstruction code for the closures and timers. Listing 7 depicts the generated snapshot code. An immediately-invoked function restores the closures to eliminate the possibility of side effects since the execution context

```
1  /*  DOM objects  */
2  ...
3  /*  The instrumented code (Listing 4) comes here */
4  ...
5  /*  Restore global variables */
6  ...
7  /*  Restore the disclosure table  */
8  (function(){
9    var $disc0 = $disc.create()
10   $disc0.$scopes.$disc0 = $disc0.$ref_index
11   $disc0["count"] = 3
12   $disc0["$ret_func"] = function () {
13     $disc0["count"] += 1
14     console.log($disc0["count"])
15   }
16 })()
17 var myCounter = ($disc.get_ref(0))["$ret_func"]
18
19 /*  Restore the wrapped timer methods */
20 disc_setTimeout(disc_setInterval(function() {
21     myCounter()
22 }, 1000), 250)
```

**Listing 7** The snapshot code serialized by Disclosure, 3.75 seconds after app loading of Listing 4.

of the function should not be referenced by another context (lines 8–15). In particular, the closures are restored by the *$disc.create* function again to generate the scope objects in the order they were created (line 9), after which the other properties are restored as well (lines 10–15). Following this, the closure is assigned to the global variable *myCounter* with the index from the disclosure table (line 17). Last, the timers are registered with the wrapper functions (lines 20–22). Suppose the serialization event occurs 3.75 s after the app is loaded. In that case, the *setInterval* method must be triggered 0.25 s after the snapshot is loaded and executed and repeatedly triggered every 1000 ms thereafter. Thus, the serialization method utilizes the *disc_setTimeout* method, which is fired only once to make up 0.25 s.

Restoration is simply achieved by transferring the snapshot file to the target device and then executing it on any web browser with the runtime library. Figure 3 presents an example scenario. For example, consider a case with four agents: a web server, proxy server, smartphone, and PC. If users wish to run a web application that is migratable later, they can use the proxy server to download the application from the web server. Then, the proxy server instruments the downloaded application and delivers it to users with the runtime library (which can be omitted if users already possess it). Suppose

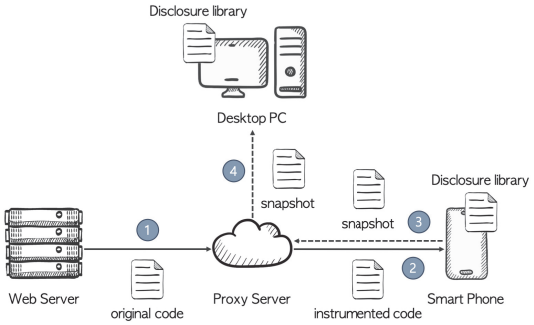



**Figure 3** One possible migration scenario.

that the users run the instrumented application on the smartphone for a while and, at some point, want to enjoy it on a larger screen. Then, they simply press the migration button to capture and transfer the snapshot to the proxy server and relay it to a desktop PC, where they can run it continuously.

Regarding the DOM tree, Disclosure creates all DOM objects as JavaScript objects by creating them as a result of executing the instrumented code. Therefore, we can capture the DOM objects as regular JavaScript objects, even for multiple migrations. Even for DOM objects created dynamically, we can know the execution context in which they are created, so if they are included in the closures or web API states, we can add them to the disclosure table with the scope object, allowing for their restoration. This works differently from Imagen [3], which uses the JsonML library to save the DOM tree [9]; it is not clearly described how to recover the link between JavaScript variables and the DOM objects referenced by them after restoring the DOM tree or how to handle dynamic DOM objects. ThingsMigrate and FlashFreeze do not support DOM migration.

## 5.4 Memory Leak and WeakMap

Currently, there is a memory leak issue with Disclosure. Since Disclosure should maintain the value and scope chain of the free variables in the disclosure table, it currently prevents garbage collection from automatically releasing the closure variables even after any variables no longer reference them. We can solve this problem by using a feature called *WeakMap* [10] in the JavaScript specification, which enables garbage collection for the elements of a WeakMap object that are not accessed for a long time (i.e., *weak references*). Unfortunately, WeakMap does not yet support the iteration for the elements, so we cannot create the disclosure table as a WeakMap

object, as we should iterate over the elements of the table to create a snapshot file. Iterable WeakMaps are currently under development [11], so we leave the solution as a future work item.

## 6 Evaluation

We evaluate Disclosure using eight Octane benchmarks [12] and four real web applications. The eight benchmarks are *Richards*, *Deltablue*, *Crypto*, *Raytrace*, *Regexp*, *Splay*, *SplayLatency*, and *NavierStokes*. Meanwhile, the web applications are *Tetris*, *Sokoban*, *Maze*, and *Emoticolor*. For the benchmarks, we measure the runtime performance, while for the web applications, we examine the overhead of multiple migrations in capturing DOM objects and timer methods. Closures are used by three web applications and three benchmarks. Moreover, we experimented with those benchmarks with no closures to confirm that Disclosure has no side effects. Timeout methods and DOM objects exist only in web applications. We conducted the experiments on the Google Chrome browser version 64, running on the Ubuntu 16.04 LTS with an Intel i7-2600 CPU 3.40 GHz and 16 GB RAM.

### 6.1 Instrumented Code Size

Code instrumentation increased the code size by 0–20%, as depicted in Table 1. The result demonstrates that the increase in code size is proportional to the number of scope objects with closures and timeout methods. If there are no closures or timeout methods, code instrumentation leaves the code intact, implying that our work had no side effects in the instrumentation phase. Therefore, the instrumented codes of *Richards*, *Crypto*, *Raytrace*, and *Splay* (*SplayLatency*) are identical to the original codes. Conversely, other benchmarks like *Deltablue*, *RegExp*, and *NavierStokes* have closures, and thus the code sizes increased up to 20%. In particular, the code size of *RegExp* greatly increased due to the numerous closure variables. Among the real applications, the code sizes of *Maze* and *Tetris* increased by 3.81% and 5.11%, respectively, because both had closures and timeout methods. *Emoticolor* had a few closures but numerous timeout methods, so the instrumentation increased the code size by about 2%. However, the code size of *Sokoban* hardly increased since it has only a few timeout methods. Meanwhile, the size of the runtime library is only around 50 KB, which contains code that allocates the *$disc* object, including the disclosure table and serialization method, and defines wrapping functions for timer methods. The instrumentation time takes about

**Table 1** Instrumentation data for eight Octane benchmarks and four web apps

| | Benchmarks | Original Code (Bytes) | Instrumented Code (Bytes) | Increase (%) | Instrumentation Time (ms) | Scope Objects | Closure Variables | Timeout Methods |
|---|---|---|---|---|---|---|---|---|
| Octane | Richards | 9076 | 9076 | 0 | 85.2 | 0 | 0 | 0 |
| | Deltablue | 15478 | 15715 | 1.53 | 110.3 | 1 | 2 | 0 |
| | Crypto | 45519 | 45519 | 0 | 236.5 | 0 | 0 | 0 |
| | Raytrace | 22248 | 22248 | 0 | 147.1 | 0 | 0 | 0 |
| | Regexp | 132929 | 146215 | 9.99 | 464.9 | 1 | 312 | 0 |
| | Splay | 6599 | 6599 | 0 | 82.3 | 0 | 0 | 0 |
| | SplayLatency | 6599 | 6599 | 0 | 82.3 | 0 | 0 | 0 |
| | NavierStokes | 11542 | 13813 | 19.68 | 109.8 | 2 | 32 | 0 |
| Web Apps | Maze | 7226 | 7501 | 3.81 | 162.3 | 1 | 1 | 4 |
| | Tetris | 25038 | 26317 | 5.11 | 121.1 | 4 | 2 | 8 |
| | Emoticolor | 16872 | 17203 | 1.96 | 88.2 | 1 | 16 | 9 |
| | Sokoban | 59121 | 59144 | 0.04 | 128.3 | 0 | 0 | 2 |

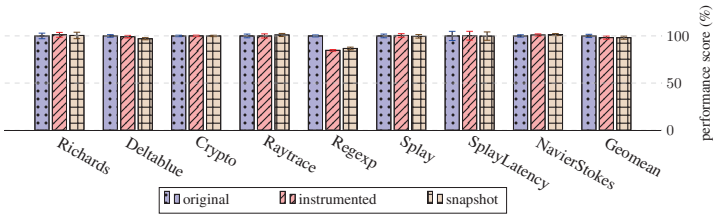



**Figure 4** Performance of the original, instrumented, and snapshot code.

82.3–464.9 ms, proportional to the original code size and the number of closures and timer methods. Since this is fast enough, instrumenting on a proxy server before loading an application would not affect the user experience.

## 6.2 Execution Performance and Runtime Memory Usage

We employed the Octane benchmark suite to evaluate the execution performance of the original, instrumented, and snapshot code. Figure 4 is the result of the benchmark scores, with the original as a basis of 100% (note that higher is better). We measured the benchmark scores 2000 times, obtained an average and standard deviation, and then checked the correctness of the result through its checksum. We did not measure the performance of web applications, as they are executed in an event-driven manner via asynchronous events.

The experimental result reveals that the average benchmark score of the instrumented codes is 2% lower than that of the original codes, and the benchmarks without closures were hardly affected. However, the benchmarks with closures such as *Deltablue*, *Regexp*, and *NavierStokes* exhibit a performance decrease of 1%, 15%, and 0%, respectively. Meanwhile, *Deltablue* and *NavierStokes* have little performance loss, compared with the number of

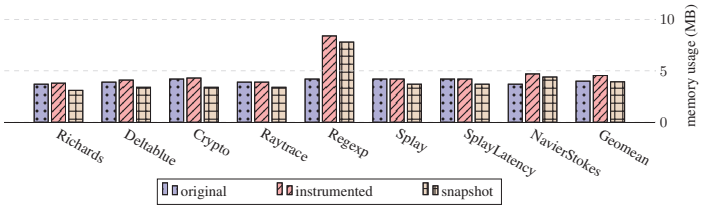



**Figure 5** Memory usage of the original, instrumented, and snapshot code.

closure variables, because they use closure variables only during initialization, primarily using the global objects thereafter. On the other hand, *Regexp* instantiates many string objects defined as closure variables and accesses them frequently during execution, thus exhibiting a higher loss.

We took a snapshot immediately after the initialization process, and the execution performance of the snapshot is generally similar to that of the instrumented code. However, unexpectedly, the snapshot performance of *Raytrace* and *NavierStokes* is slightly higher than that of the instrumented and original codes. We found that this occurs when the snapshot includes the execution result of a non-trivial initialization process, so bypassing initialization yields some benefit. Moreover, the snapshot performance of *RegExp* is higher than that of the instrumented code but lower than that of the original code due to many closure variables needing to be recovered.

The memory usage of each benchmark is depicted in Figure 5. The memory usage of the instrumented code is larger than that of the original code by approximately 14% due to the newly generated scope objects and increased code with the runtime library. *Regexp* and *NavierStokes*, which have many closure variables, use much more memory than the original code. In particular, the memory usage of *Regexp* increased almost two-fold compared with the original. This is due to a memory leak in the disclosure table, as mentioned in Section 5.4.

Noteworthy is that the memory usage of the snapshot is always smaller than that of the instrumented code and is, in some cases, even smaller than that of the originals. This is also due to the elimination of the initialization process.

## 6.3 Multiple Migrations

We migrated our web applications multiple times in the middle of the execution and confirmed that we could restore and proceed with the execution. In addition, we checked whether Disclosure itself causes any side effects on

runtime memory usage or the snapshot size. The applications were executed by events after the initialization process ended, with the global context being terminated. The events were issued by timers or user interactions such as button clicks. For example, *Maze* repeatedly generated random walls to complete a maze, and *Tetris* periodically generated blocks. Thus, both increased memory usage continuously up to some point, namely when the maze was completed or when the game ended, and we migrated at this point. On the other hand, *Emoticolor* generated buttons with RGB color codes randomly during app loading, and *Sokoban* generated a map and waited for a keyboard signal, so we migrated immediately after the app loaded without needing to wait.

We performed five migrations for each app by iterating the process of (1) taking a snapshot, (2) migrating, and (3) restoring execution five times. When we measured the snapshot code size during each of the five migrations, we found it to be the same for all apps. Furthermore, we observed no differences in the runtime memory usage during the five migrations. These findings imply that Disclosure has no side effects.

### 6.4 Comparison with ThingsMigrate

We measured the runtime performance of the instrumented code generated by Disclosure and ThingsMigrate using the Octane benchmarks, save for *RegExp*, as ThingsMigrate did not cover it [6]. Instead, ThingsMigrate included *Factorial* to evaluate computation-intensive algorithms, so we included that as well. The experiment was conducted on Node.js v10.15.3, and the performance scores were measured 100 times. Figure 6 presents the result, in which Disclosure significantly improves the performance, around 30 times faster than ThingsMigrate. The low performance of ThingsMigrate is primarily due to mirroring statements. The performance of *Raytrace*, *NavierStokes*,

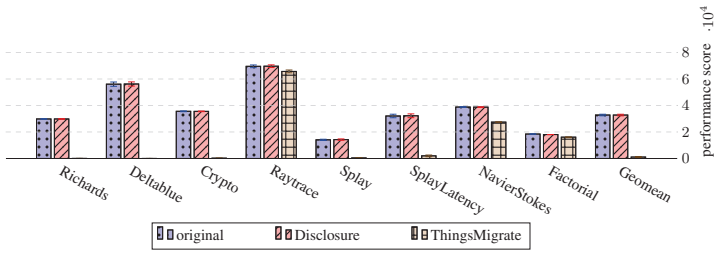



**Figure 6** Performance of original, disclosure, and ThingsMigrate code.

```
1  var $disc = function() {
2      let disc_table = []
3      return function() {
4          this.create = function() {
5              ...
6          }
7          this.serialize(key) = function() {
8              if (key)
9               disc_table =
10              disc_table.map(e=>encrypt(e, key))
11             return JSON.stringify(disc_table) }
12         this.decrypt(key) = function() {
13             disc_table =
14             disc_table.map(e=>decrypt(e, key)) }
15     }
16 } ()
```

**Listing 8** The initialization code for disclosure table.

and *Factorial* is not much lower since they do not have many mirroring statements. Thus, it is clear that the mirroring statements are a serious performance bottleneck, which Disclosure can decently alleviate.

## 7 Secure Migration

In JavaScript, closures are primarily used to encapsulate variables. However, the snapshot file can reveal the values of the closure variables as in Listing 3 or Listing 7, affecting security since it stores raw data in the form of JavaScript code. Fortunately, a Disclosure can enhance security in light of that it implements the disclosure table itself as a closure. That is, when we initialize the $disc library, we declare the disclosure table and have the inner function create() add a new element (for a closure variable of the original code) to the table as depicted in Listing 5. Therefore, we can create encrypted snapshot files without knowing the elements within the disclosure table. As depicted in Listing 8, we can encrypt the disclosure table by passing an encryption key to the serialize() function to obtain the encrypted snapshot file (lines 7–11). Then, transfer the encrypted snapshot file to the target device and restore it. Finally, call the decryption method to decrypt the disclosure table by passing a decryption key to decrypt each element found in the disclosure table (lines 12–14). We name this secure migration, and it would ensure the snapshot file does not expose any sensitive data while restoring the original disclosure table wrapped by a closure, thereby enhancing security further.

## 8 Related Work

The snapshot can be applied to other optimizations, such as computation offloading or loading time acceleration. Computation offloading is a technique that offloads a heavy workload to a server with sufficient computing resources to handle the task more efficiently. Disclosure can capture a snapshot of a web application by serializing the entire application state [13] or the web worker state that handles a heavy task [14, 15]. In addition, since JavaScript applications typically exploit heavy frameworks that create many objects at loading time, restoring a snapshot is more efficient than loading and initializing the source code [16, 17], which is discussed in Section 6.2.

Meanwhile, there are several applications in blockchain, such as UniSwap [18], Compound [19], and ENS (Ethereum Name Service) [20]. These are related to decentralized finance (Defi), often providing standalone applications using blockchain as a database without a web server. Therefore, these applications are sometimes hosted over P2P networks like IPFS [21], Filecoin [22], and Arweave [23]. Furthermore, security is critical in Defi, so hiding the state during migration is very important. So, these applications will likely be making heavy use of liquid computing with secure migration in the future when blockchain becomes a widely used infrastructure.

## 9 Conclusion

In this paper, we propose Disclosure, a new instrumentation-based migration technique for web applications based on a disclosure table, which keeps the instrumented code from seriously slowing down while allowing for multiple migrations. Also, Disclosure can enhance the security of the snapshot file and fully migrate the DOM tree, unlike in previous works. Our experimental results showed that the benchmark performance was a much better result than previous work. These advantages enable standalone applications, such as game and blockchain applications, to run across multiple devices without a web server. Finally, supporting features of ECMAScript6 [24] and covering a memory leak are left as future work.

## Acknowledgement

This work is an extended version of “Disclosure: Efficient Instrumentation-Based Web App Migration for Liquid Computing” published at International Conference on Web Engineering, 2022 [25], supported by Institute of

Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00180, 10%) and (No. 2021-0-00136, 10%), and by the ITRC (Information Technology Research Center) support program (IITP-2021-0-01835, 80%) supervised by the IITP.

## References

[1] Piotr Sroczkowski. 100 most popular languages on github in 2019. https://brainhub.eu/blog/most-popular-languages-on-github, 2020.

[2] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 105–110. ACM, 2011.

[3] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22nd international conference on World Wide Web (WWW)*, pages 815–826. ACM, 2013.

[4] JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. Migration of web applications with seamless execution. In *2015 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM, 2015.

[5] Jin-woo Kwon and Soo-Mook Moon. Web application migration with closure reconstruction. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pages 133–142, 2017.

[6] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[7] T. Mikkonen, K. Systa, and C. Pautasso. Towards liquid web applications. In *Proceedings of the 2015 International Conference on Web Engineering (ICWE).*, pages 134–143. Springer, 2015.

[8] Jonathan Van der Cruysse, Lode Hoste, and Wolfgang Van Raemdonck. Flashfreeze: low-overhead javascript instrumentation for function serialization. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, pages 31–39, 2019.

[9] JsonML. http://www.jsonml.org/.
[10] MDN web docs: Weakmap. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap.
[11] Iterable maps. https://github.com/tc39/proposal-weakrefs#iterable-weakmaps.
[12] Octane Javascript benchmark. http://chromium.github.io/octane.
[13] Hyuk-Jin Jeong, Inchang Jeong, and Soo-Mook Moon. Dynamic offloading of web application execution using snapshot. *ACM Trans. Web*, 14(4), jul 2020.
[14] Maciej Zbierski and Przemyslaw Makosiej. Bring the cloud to your mobile: Transparent offloading of html5 web workers. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 198–203, 2014.
[15] Hyuk-Jin Jeong, Chang Hyun Shin, Kwang Yong Shin, Hyeon-Jae Lee, and Soo-Mook Moon. Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 38–49, 2019.
[16] JinSeok Oh and Soo-Mook Moon. Snapshot-based loading-time acceleration for web applications. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 179–189. IEEE, 2015.
[17] Google. Custom snapshot. https://v8.dev/blog/custom-startup-snapshots, 2018.
[18] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.*, 2021.
[19] Robert Leshner and Geoffrey Hayes. Compound: The money market protocol. *White Paper*, 2019.
[20] Ethereum name service. https://ens.domains/.
[21] Ipfs powers the distributed web. https://ipfs.tech/.
[22] Protocol Labs. Filecoin: A decentralized storage network. Retrieved from: https://filecoin.io/filecoin.pdf, 2017.
[23] Sam Williams and Will Jones. Archain: An open, irrevocable, unforgeable and uncensorable archive for the internet. DOI: https://www.arweave.org/whitepaper.pdf, 2017.
[24] Ecma-262 edition 6.0 – ecmascript language specification. https://262.ecma-international.org/6.0/, 2015.
[25] Jae-Yun Kim and Soo-Mook Moon. Disclosure: Efficient instrumentation-based web app migration for liquid computing.

In *International Conference on Web Engineering*, pages 132–147. Springer, 2022.

[26] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[27] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly, 2017.

[28] Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[29] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 105–110, 2011.

[30] Jae-Yun Kim, Hyeon-Jae Lee, and Soo-Mook Moon. Fast snapshot migration using static code instrumentation: work-in-progress. In *Proceedings of the 15th International Conference on Embedded Software*. ACM, 2018.

[31] A. Gallidabino and C. Pautasso. Maturity model for liquid web architectures. In *Proceedings of the 2017 International Conference on Web Engineering (ICWE).*, pages 206–224. Springer, 2017.

[32] Kumseok Jung, Julien Gascon-Samson, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript internet-of-things applications. *Softw: Pract Exper.*, 51, 2021.

[33] Jin-woo Kwon, JinSeok Oh, InChang Jeong, and Soo-Mook Moon. Framework separated migration for web applications. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10. IEEE, 2015.

[34] Hyuk-Jin Jeong and Soo-Mook Moon. Offloading of web application computations: A snapshot-based approach. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 90–97. IEEE, 2015.

[35] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5), 2017.

[36] Timothy J Berners-Lee. Information management: A proposal. Technical report, 1989.

[37] Wikipedia. Closure (computer programming) – Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Closure%20(computer%20programming)&oldid=886596565, 2020.

[38] Ariya Hidayat et al. Esprima. http://esprima.org, 2020.
[39] Yusuke Suzuki et al. Esprima. https://github.com/estools/estraverse, 2020.
[40] Yusuke Suzuki et al. escodegen. https://github.com/estools/escodegen, 2020.
[41] Mozilla Developer Network. Run-to-completion. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion, 2020.
[42] Node.js. https://nodejs.org/, 2020.
[43] LG. Webos open source edition. http://webososе.org/, 2018.
[44] Samsung. Tizen. https://tizen.org/, 2020.
[45] Google. Chrome v8 javascript engine. https://v8.dev.
[46] Ecma-262 edition 5.1 – ecmascript language specification. https://262.ecma-international.org/5.1/, 2011.
[47] Google. New multi-screen world: Understanding cross-platform consumer behavior. https://services.google.com/fh/files/misc/multiscreenworld_final.pdf, 2012.
[48] Thingsjs. https://github.com/DependableSystemsLab/ThingsJS/tree/e46dae718b9822800bbd623bb95cc6de2c51d564, 2020.
[49] Mdn web docs: Property accessors. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors, 2020.
[50] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.
[51] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.
[52] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283. ACM, 1996.
[53] Olivier Danvy and Lasse R Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174. ACM, 2001.
[54] JiHwan Yeo, JinSeok Oh, and Soo-Mook Moon. Accelerating web application loading with snapshot of event and dom handling. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 111–121, 2019.

[55] JiHwan Yeo, ChangHyun Shin, and Soo-Mook Moon. Snapshot-based loading acceleration of web apps with nondeterministic javascript execution. In *Proceedings of the The 2019 World Wide Web Conference (WWW'19)*, 2019.

[56] Yong-Hwan Yoo and Soo-Mook Moon. Snapshot-based migration of es6 javascript. In *Proceedings of the The 2021 International Conference on Web Engineering (ICWE 2021)*, 2021.

[57] Hyuk-Jin Jeong, InChang Jeong, Hyeon-Jae Lee, and Soo-Mook Moon. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1492–1499. IEEE, 2018.

[58] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411, 2018.

## Biographies

**Jae-Yun Kim** received his bachelor's degree in electrical and computer engineering from Seoul National University in 2015. He is a Ph.D. candidate studying at Virtual Machine and Optimization (VM&O) Laboratory in the Electrical and Computer Engineering Department at Seoul National University. He is currently a Founder and CEO of Superblock Co., Ltd., designing and implementing a new blockchain network focusing on lightweight node clients.

**Soo-Mook Moon** received his Ph.D. at the University of Maryland, College Park, in 1993. During 1992–1993, he worked at IBM Thomas J. Watson Research Center, where he developed the IBM VLIW compiler. During 1993–1994, he was a software design engineer at the Hewlett-Packard Company in California Language Lab, where he contributed to developing an optimizing compiler for the PA-RISC CPUs. Since 1994, he has been with the faculty of the Seoul National University in the Department of Electrical and Computer Engineering, where he is now a full professor. Now, he leads the Virtual Machine and Optimization (VM&O) Laboratory in the Department of Electrical and Computer Engineering at Seoul National University. Research areas of VM&O Lab are blockchains, federated learning, compiler optimizations, language virtual machines, and web platform optimization.